Conversational Interfaces: A Domain-Independent Architecture for Task-Oriented Dialogues

Alexander Gruenstein
alexgru@stanfordalumni.org

M.S. Project
Symbolic Systems Program
Stanford University

December 12, 2002

Abstract

This paper motivates and describes a generic framework for dialogue-enabling intelligent agents and devices for *task-oriented dialogue*. The interface is designed to be a mechanism by which a dialogue front-end can quickly and easily be adapted for use with a wide range of devices or agents. The *Conversational Intelligence* requisite for participating in a large range of important task-oriented dialogues is identified and decomposed in a modular, device-independent fashion and a specialized *recipe scripting language* is implemented to encode device-specific information. The recipes in the recipe library compiled from this scripting language are instantiated at run time into *Activities*, which may be executed by the device (and jointly, by the human operator). In addition, a novel *Constraint Management System* is implemented in order to exploit the features of natural language which allow humans to naturally expand and restrict the permissible sets of parameters that a particular activity may take on.

Contents

1 Introduction

Natural language, whether spoken or typed, is an emerging means of interacting fruitfully with computer systems. The ability to order movie tickets or find out directions over the telephone using voice recognition technology, to name two examples, is emerging as mainstream technology. At the same time, more and more sophisticated semi-autonomous computer controlled agents/devices are in development or have been developed. For example, mobile robots have become particularly robust with regard to low-level issues like localization and obstacle avoidance [KBM98]; moreover, mobile robots have started to become extremely cheap, costing just a few thousand dollars, and are widely available [act]. One of the original such devices was built by the FLAKEY project at SRI (see, eg [SKR95]), which produced a semi-autonomous office robot that could navigate through an office to do tasks such as delivering items. Current work includes the WITAS project [DGK$^+$00] which endeavors to create an autonomous helicopter – an unmanned aerial vehicle – capable of planning efficient routes, identifying objects on the ground, and following moving objects. In addition, NASA is developing a Personal Satellite Assistant (PSA) that moves autonomously about the shuttle or the space station to assist the crew [RHJ00]. And the long-running Honda humanoid robot project is developing a humanoid robot named ASIMO that can navigate the home and perform simple chores [hon].

Related projects are those involved in dialogue-enabling existing devices in the home, office, and automobile. The aim is to allow humans to control devices like telephones, televisions, radios, cd players, and VCRs with spoken natural language. Such projects are being pursued, for example, at Telia [LRGB01], COLLAGEN [RSL01], Bosch and SmartKom [LBPA02]. These groups hope to build natural language interfaces which can be integrated into existing technology, rather than devices which are still under development. By doing so, they hope to change the way that humans interact with electronic devices by making it easier for humans to interact with them using natural, spoken language.

Running the gamut from the more intelligent devices to the relatively "dumb" devices is the idea that these devices are *engaged in activities in a dynamic environment.* These devices, to one extent or another, make plans for action and then execute them – whether

it be a plan to follow a car in the case of the WITAS project, or a plan about how to set the time of the VCR in COLLAGEN. Moreover, for each activity there may be many *parameters* that need to be specified by the human operator, inferred by the system, or even randomly chosen. For instance: a helicopter needs to know where to go, at which altitude and speed to fly, and possibly even myriad other flight parameters like pitch, yaw, or roll; a VCR should know what time to begin taping a show, when to end, what channel it's on, and at what quality to record; and a humanoid robot might need to understand how careful to be, the volume at which to speak, or at what speed to move. The point of natural-language enabling these devices should be to make them easier to control. By producing language, they should be able to effectively communicate their current *state* to the human operator in terms which will make sense to the human, negotiate with the operator about the values of parameters, and answer questions about their state in a natural manner. Similarly, by understanding language, they should be able to give the operator the means to easily modify their state or enquire further about its details in a natural manner. Moreover, language should allow them to be proactive in a natural way: they should initiate information-seeking or clarification dialogues when necessary, without forcing the user to navigate complex menu systems on a screen or understand how to write programming code. Finally, they should be able to participate naturally in *joint activities* (see [Cla96]) in which *both* the human operator and the intelligent device *collaborate* in order to bring about a desired outcome. These activities provide and importance context by which utterances in a conversation should be understood.

Most or all of these advantages that arise from using natural language are desirable across a wide range of devices. Moreover, many of these advantages don't arise from the underlying intelligence of the device itself; rather, they come from a different sort of intelligence: Conversational Intelligence (CI) [LGP02]. Conversational Intelligence is knowledge about how and why conversations occur between agents, and how to effectively participate in conversations. While it's true that without an underlying intelligence, it's difficult to have an interesting conversation *about* anything very interesting, it's not the case that with knowledge and intelligence come the ability to communicate effectively. Some knowledge, including for example knowledge about when it's appropriate to speak, how and when to interrupt someone who is speaking, how the context provided by past utterances should

be used to interpret new ones, and when it's important to mention particular changes in the state of the world are just a few examples of Conversational Intelligence.

The project discussed in this paper revolves around the intertwined goals of actually defining what's involved in CI and implementing such knowledge using a computational system in a device-independent, modular manner. Specifically, introduced here is a device-independent architecture for building an interface to the CSLI Dialogue Manager [LGP02] such that its conversational front-end can be quickly and easily interfaced to a wide-range of devices. Moreover, this interface was designed to provide its own conversationally intelligent mechanisms which may be harnessed by the dialogue manager in support of more complex, yet natural dialogues with the devices. For example, device-independent support is provided for *constraint* dialogues in which natural language is used to restrict and expand the permissible sets of values of the parameters on particular activities defined by a device (see section 7). Such dialogues allow human operators to change the overarching parameters which control a device in an intuitive and smooth manner.

In section 2, will discuss previous and current work in dialogue systems that is relevant to the project at hand, focusing mainly on those that deal with either commanding devices or coordinating actions of human agents. I will also study some of the theories that have emerged regarding how rational agents operate and communicate, as these theories shed light on how, why, and when a rational agent should communicate. Next, I will look at intelligent devices that have actually been designed, and the sorts of constructs which have been used to control them. I will then note the importance of accurately communicating the state of these devices to the human operator, so that mode confusion can be avoided.

In section 3, I will then show how my project is relevant to this research, and how it naturally extends much of the work. Then, I'll delve into the depths of the project. In section 4, I'll present the formalism of the Activity Tree, developed to represent that current state of the agent or device with which the operator is communicating. In sections 5 and 6, I'll discuss the special language I've created to interface devices to dialogue systems. Then in section 7, I'll discuss the constraint management system developed as part of this project, which ranges over the activity representation developed in the previous sections. Finally, in section 8, I'll discuss how the formalism is

implemented and the interface between the dialogue system and the device is achieved. There, I'll give a brief description of the functioning of the CSLI Dialogue Manager; though it is important to note that the architecture I've implemented here could be extended to fit in with other dialogue managers, built on different theoretical underpinnings. The point of my project is not to manage the intricacies of spoken-language dialogue, but to provide resources by which a dialogue manager can facilitate meaningful dialogue with a wide-range of devices. I will develop algorithms by which a dialogue manager should interact with the facilities discussed in this paper.

2 Previous and Current Work

In this section, I will delve into several areas of research in order to highlight the myriad useful ideas that have emerged, as well as to show where this research needs extension and implementation. This background will show how the framework discussed in this paper fits into large areas of research.

2.1 Dialogue Systems

There is a wide range of dialogue systems that have been commercially deployed, have been developed for research purposes, are currently being developed, or are planned to be developed according to theoretical work in progress. Such systems range in complexity depending on the difficulty of the problem for which they are designed. I'll discuss here a range of such systems and the tasks for which they have been designed; a useful and often parallel discussion of the range of developed dialogue systems appears in [ABD+01].

2.1.1 Slot/Form-Filling Dialogue Systems

There has been a large amount of work on so called slot-filling dialogue systems. Such dialogue systems are useful in domains where certain bits of information need to be elicited from the user, resulting in a set of *slots* being filled, which are usually used to make a database query or update. For instance, when designing an automated airline reservation system, dialogue designers have often thought in terms of the specific bits of information that the user must supply in order for the system to do a database search for available flights that match

this set of criteria. Such a system might have the following slots that need to be filled, where for each there is a domain of allowable values:

- The departure city
- The arrival city
- Date on which to travel
- Time at which to travel

In order to fill in these slots, dialogue systems generally use some combination of *user initiative* and *system initiative* (a combination referred to as *mixed-initiative*). This means that the user may provide the values for some subset of the slots in a single utterance and then the dialogue system can ask follow up question to elicit the rest of the required information. This stands in opposition to earlier *finite-state* dialogue systems, which required that a series of questions be asked and answered in a specific order so that all of the slots might be filled in. For example, slot-filling dialogue systems can often understand utterances like *I'd like to fly from San Francisco to London* which provide some of the necessary information required to make an airline reservation, but not all of it. The system will follow up with information-seeking questions when the user fails to fill in all the necessary slots. For example, in response to the above utterance, a system might respond with *Okay. When would you like to depart?*.

In the most straightforward instantiations of such systems, the human user must fill all of the slots before proceeding – though the slots may be filled in any order. Such an architecture has been used to build, for example, airline reservation systems (*e.g.* [SP00]) and train timetable systems (*e.g.* [SdOB99]). It has also been commercialized by companies like *Nuance* and *Tellme* who build customized applications for clients like banks, telephone companies, and airlines.

Form-filling dialogue systems, then, conceptualize information-seeking dialogue in terms of a mapping from user-utterances to values for slots. Once the requisite slots are filled, the system can take some sort of action – for instance, making an airline reservation [SP00]. Conversational Intelligence is demonstrated to the extent that it has a *strategy* for eliciting information that the back-end of the system needs from the user in order to take some action. It is clear, however, that a simple form-filling model is not sufficient for controlling intelligent agents in complex environments: there is no mechanism, for instance, to answer questions about the state of the device, or *why* the device is doing

a particular action, since these things are not modeled. On the other hand, form-filling provides a good model for a means of obtaining values for a set of parameters, which is highly relevant to some aspects of controlling intelligent devices.

2.1.2 Practical Dialogue Systems

In [ABD+01] the authors identify a type of dialogue which they refer to as *practical dialogue*. They define practical dialogue as dialogue which "may involve executing and monitoring operations in a dynamically changing world" ([ABD+01]:3). As opposed to the types of dialogue systems discussed in the previous section, dialogue systems which are designed to work at such a level generally facilitate interaction with devices in a real-world or simulated environment, with the goal of accomplishing some specific task (in contrast to simply doing a database lookup, for example). Allen, *et al*, claim that while such dialogues are complex, they don't require full-human competence to understand and participate in. Indeed, it is apparent that a system designed to handle such dialogues could probably function pretty well without understanding how a metaphor, for example, functions in language.

There are many current research projects which are involved in trying to build dialogue systems which function at this "practical" level. Many of them are trying to dialogue-enable existing devices in the home, office, and/or automobile (*e.g.* [LRGB01], [LBPA02], [RSL01]). Such projects often are pursuing fruitful ways to endow capabilities like clarification sub-dialogues and anaphora resolution across a diverse set of devices. For example, such systems try to generically enable dialogues like the following:

(1) a. O: Turn on the light.
S: Which light should I turn on?
O: The one in the kitchen.
S: The light in the kitchen is now on.

b. O: Is the light in the kitchen on?
S: Yes.
O: Turn it off.
S: The light in the kitchen is now off.

In the above dialogue, the lights are referred to with *anaphoric* expressions such as *the one in the kitchen* and *Turn it off* – an ability

above and beyond that supplied by the simpler form-filling dialogues. Moreover, rather than simply doing static-database queries and reporting the results, the dialogue system must be able to monitor a world in which the dialogue may cause changes (here, the lights turn off and on). Such dialogue systems must carry with them some notion of how the devices they control operate – they must understand that electronic devices can be turned on and off, radios can be turned to a particular station and only dimmable lights can be dimmed to particular intensities. In order to address these issues, in [LRGB01] for example, a plug-and-play system is designed in which new devices with new capabilities can be added to the system easily by loading in the linguistic resources for controlling and querying a device, the abilities of the device, and the code used to interface to the device on the fly.

In this domain, it is evident that the Conversational Intelligence of the device can begin to be separated from its function. By separating linguistic and ontological knowledge so that both can be applied across many similar devices in order to enable dialogue phenomena like anaphora resolution and clarification sub-dialogues, and in order to capture commonalities like the fact that electronic devices may be turned on and off, researchers create generic dialogue systems which can be used to control diverse sets of devices.

Perhaps a step up in complexity from such "dumb" devices are projects that are concerned specifically with dialogue-enabling the more intelligent devices that are under development for the future. At NASA, for example, a dialogue system has been created for the Personal Satellite Assistant (PSA) under development there [RHJ00], a small mobile robot for use in the space station. The dialogue manager gives astronauts a means of giving orders to the PSA such as *Measure the temperature at the captain's seat* and allows the PSA to disambiguate orders such as *Open the hatch* when there are multiple hatches. While the project has produced a workable dialogue manager, no general results have been reported about the sorts of devices that the dialogue manager would be able to control, or even if it would be able to control devices besides the PSA.

At CSLI, I am involved as part of the Computational Semantics Lab in the WITAS project [LGP02], which has as its goal to dialogue-enable an autonomous helicopter currently under development [DGK$^+$00]. In order to dialogue enable this device, we have sought to define what knowledge constitutes the *conversational intel-*

ligence that is needed by a wide-range of task-oriented autonomous devices in order to effectively communicate. This paper will describe the interface to the robot helicopter that the dialogue manager being developed at CSLI uses in order to behave with conversational intelligence in such areas as: when to make an utterance, when to ask a question, how to set devices' parameters, and so on. It is our aim to make a generic dialogue system that can be straightforwardly specialized to dialogue-enable the large range of autonomous devices that may be developed in the future which might be quite different from the helicopter with which we are currently working.

As was touched on briefly in section 2.1.1, dialogue systems for controlling such devices are inherently more complex than form-filling dialogue systems because they must model, to some extent or another, the state of the world. Specifically, they must be able to model the current *joint activities* in which the intelligent agent and the human are involved.

2.2 Rational Agents and Plan Recognition

In order for a dialogue system to work with intelligent devices, it must be capable of understanding the sorts of plans that a user wishes to carry out with the device. There has been a large amount of research into understanding how rational agents (such as humans) conceptualize plans and communicate about them. This has bearing on the project at hand, both because the dialogue system ought to be able to facilitate the understanding of the intentions of the human operator and because it should be able to communicate the plans formed by the device to the human; if the actions of the intelligent device are to be understood as rational by the human operator, then the dialogue manager must be able to coherently communicate them to the human in terms that will make sense to him or her. Inferring the plans of rational agents based on what and how they communicate is called *plan recognition*.

Critical to understanding what it means to have a plan is a distinction drawn in [Pol90]. Here, Pollack makes a crucial distinction between *plans* and *recipes*: a *recipe-for-action* is a recipe by which rational agents formulate plans, and a plan is an instantiation of a particular recipe-for-action. A plan is part of *complex mental state* of a conversational agent, whereas a recipe is a more abstract notion – it composes part of the *recipe library* from which an agent might

choose a recipe to instantiate into an actual plan in a particular situation. When an agent wishes to achieve a goal, he/she/it instantiates a recipe into a plan by which the goal may be achieved.

A plan describes a means of accomplishing a goal or completing an action. It does this by defining a set of steps – steps may appear in sequence to one another, and sets of steps may recursively be substeps of a different step. In [Bra90], Bratman points out that the ability to break up a plan into sub-plans, and still further into smaller steps, is a pragmatic one because such subdivisions allow the agent to defer planning the details of a sub-plan until later, when the state of the world may have changed in unexpected ways forcing the agent to discard or replace the sub-plan. In [Pol90] and further fleshed out in [ASF$^+$95] is a particular notion of the different ways in which steps in a plan can stand in relation to one another. In particular, the relationships given below are defined. I've added question-answer pairs to each relation which are meant to illustrate the relations:

1. **Affect**: action to state
   Q: Why are you flying to the tower?
   A: In order to be at the tower.
2. **Enablement**: state to action
   Q: Why do you want to be at the tower?
   A: So I can drop medical supplies there.
3. **Generation**: event to event (illustrated with the *by* locution)
   Q: How will you put out the fire at the school?
   A: I will put out the fire at the school *by* flying to the lake, picking up water there, flying to the school, and dropping the water on the fire there.
4. **Justification**: state to state
   Q: Why do you have to be at the tower?
   A: Because you want medical supplies there.

While this defines the relationship between particular segments of a plan it does not define precisely the notion of what it means for a conversational agent to *have a plan*. Pollack (in [Pol90]) gives the following definition: An agent A has a plan to do β that consists in doing some set of acts Π, providing that:

1. A believes that he can execute each act in Π.
2. A believes that executing the acts in Π will entail the performance of β.

3. A believes that each act in Π plays a role in his plan.
4. A intends to execute each act in Π.
5. A intends to execute Π as a way of doing β.
6. A intends each act in Π to play a role in his plan.

The fact that agents have plans which they talk about, even though these plans are not actually plans to communicate, is an important distinction made by researchers who have sought to describe ways to do plan recognition. For instance, in [LA90], the authors define *domain plans* as plans which might be performed in a particular domain, and describe those plans in a STRIPS-style formalism. The notion of agents "having plans" and collaborating with one another about them is also developed by Groz, Sidner, and others in a number of papers (for example: [GS90], [GK96], [GK98]). The result of their research is the definition of a *SharedPlan*. *SharedPlan*s can be used to define the plans that rational agents make with one another; indeed, they can be used to determine when agents have successfully communicated to one another that they intend a particular *plan*.

Attempts have been made to apply the *SharedPlan* Model to dialogue systems (*e.g.* [Loc94]). Most recently, this has been done by the Sidner's COLLAGEN project (overview in [RSL01]). In particular, it is applied mainly to tutorial dialogues in which the computer helps the human user to get through a series of steps involved in operating a device ([RLR$^+$02]) – for example, programming a VCR. While COLLAGEN purports to use a *SharedPlan* model – COLLAGEN is really only a *partial* implementation of the *SharedPlan* architecture at the moment. Indeed, while *SharedPlan* affords a relatively complex means of defining the relationships among the parts of an instantiated plan, the *recipe-trees* in COLLAGEN are not nearly so rich; they consist mainly of actions which have been hierarchically decomposed. Moreover, this application is focused not on controlling autonomous devices in dynamic environments, but on helping human users control relatively simple electronic devices. As such, while COLLAGEN is helpful to our purposes in that it helps us to understand one way in which plans for action have been modeled, the types of plans it is capable of modeling are not as complex as those that more complex devices actually form, as will be discussed below.

Plan recognition has also been used by dialogue systems – especially in the TRAINS (and subsequently TRIPS) system at the University of Rochester. The most recent work on plan recognition from

that project is described in [Bla01]. The plan recognition system is focused mainly on inferring user plans either bottom up (actions first) or top down (goals first), and it has been recently enhanced to model synergies that can arise from plans being *interleaved*. Interleaved plans are those in which one action is part of several plans.

As a final note, I'd like to draw attention to the fact that in [TA94], the authors discuss an important distinction that can be drawn among the ways that a plan (or sub-plan) can culminate "successfully." Specifically, three results may be obtained:

- Successful Completion (all actions performed and goal met)
- Action Completion (all actions performed)
- Goal Satisfaction (goals achieved)

We can consider the three cases by looking at an example: consider that I have the goal to have a clean kitchen floor and my plan to achieve this goal is to first sweep the floor and then mop it. In the first case, I go ahead and sweep the floor and then mop it and the floor becomes clean – I've done all the actions in my plan and achieved my goal. In the second case, suppose that I sweep the floor and then mop it, but there's a tough stain that just won't come out; in this case, I've performed all of the actions in my plan, but my goal of having a clean kitchen floor doesn't obtain. Finally, suppose that I first sweep the floor, and then I go to the other room to look for the mop where I'm delayed by an important phone call. In the meantime, Joe notices that the kitchen floor needs a bit of cleaning, and so he takes a rag and cleans it by hand. When I return with the mop, I discover that the floor is clean even though I didn't have to mop it (and, indeed, nobody actually *mopped* it) – in this case, my goal has been achieved even though I didn't complete all of the actions in my plan.

This distinction is an important one, because it is often the case when people give commands that they don't often care about how a goal is accomplished, but just that it *is* accomplished in one way or another. On the other hand, sometimes it matters a great deal the exact way in which a goal is accomplished.

2.3 Mobile Robot Control Systems

Here, I'll consider one formalism for controlling mobile robots which has actually been fielded in many systems, and is now embodied in software which is shipped with many mobile robots in order to

control them: PRS-LITE (see, *e.g.*, [Mye96]). PRS-LITE is a re-implementation of SRI's Procedural Reasoning System (PRS) formalism streamlined for controlling mobile robots. It is now realized in the COLBERT programming language, which is part of the Saphira system, written at SRI and distributed now by ActivMedia Robotics with the robots it sells. If we see robots as representing rational agents, then we should be able to view PRS-LITE as a formalism used to actually represent the robot's plans for action – as such, we can come to understand the "mental representation" that many robots currently use in order to "have a plan." This "mental representation" has been driven by the need to effectively and efficiently control mobile robots, rather than any theoretical underpinnings of how a rational agent *ought* to behave, as the above planning formalisms have attempted to do. Understanding this is critical to my project, because if humans are to interact with intelligent devices as though they are rational agents, then we must be able to find a way to mesh the types of representations that humans use and the ones that are common to robots, if the agents have any hope of effectively communicating. Indeed, my project can be seen in terms of creating a layer by which humans and robots can interpret the intentions behind the actions and communicative attempts of one another.

PRS-LITE attempts to support the following characteristics which the authors of [Mye96] assert are requisite for controlling such mobile robots:

1. Both discrete and continuous processes
2. Concurrent activities
3. Both goal-driven and event-driven operation
4. An external observer should be able to understand the *intent* underlying the robot's action.

In addition, the authors define goal semantics which support **atomic** and **continuous** processes – that is, ones which can be used as sequential building blocks and ones which run as ongoing processes, without particular goals.

The representational basis that PRS-LITE uses is called an *activity schema*. Where an activity schema is decomposed as follows:

- It is an ordered list of goal-sets
- where a goal set is one or more goal statements ("goals") in an ordered sequence

- where a goal is a goal operator applied to a list of arguments

Goals can be decomposed *hierarchically*. Specifically, Goals are either Action Goals (Test, Execute, =, Wait-for, Intend, Unintend) or Sequence Goals (If, And, Split, Goto).

The end result is that goals are not simply hierarchically decomposed, rather, a hierarchically decomposed goal can also give rise (via split) to other trees – yielding a *forest* of actions which are currently being executed by the robot. Also, the *If* and *Goto* allow for the ability to skip over certain steps. This allows for a large amount of flexibility in the relationship between processes; however it makes it difficult for an external observer to understand the *intent* of the autonomous system. The instantiated "plan" that the system has at any one moment is simply a set of activities, some of which stand in an activity-sub-activity relationship and some of which are concurrent. There is little explicit explanation of exactly how and why the set of activities the system is running at any one time are linked together.

In typical PRS-LITE systems, there are many concurrent goals being pursued at any one time. Indeed, not only are there many concurrent tasks, but the relationships between these tasks are often abstruse. Global variables can be shared between tasks, and some tasks will wait for other tasks to set particular variables before they proceed. Given such a complex network, it is often difficult for people who monitor the system to understand exactly what the system is doing, what it intends to do, and why.

2.4 Mode Confusion in Complex Systems

Research into human understanding of complex software systems has contributed some useful insights to modeling activities as well. In [Lev00], the point is made that when activities are decomposed into smaller chunks, at each level of the hierarchy there can be observed *emergent characteristics*. On page 6, *emergence* is defined as follows:

> "Emergence – at any given level of complexity, some properties characteristic of that level (emergent at that level) are irreducible. Such properties do not exist at lower levels in the sense that they are meaningless in the language appropriate to those levels. For example, the shape of an apple, although eventually explainable in terms of the cells of the apple, has no meaning at that lower level of

description."

A similar sort of example can be found in the mobile robot domain if we consider the task of *patrol* – which, in its simplist form, consists of going back and forth between two points. At the lower level of *goto*, there is no way to explain the concept of *patrol*.

In [BL01] the authors note how important for safety it is that humans have a good understanding of how automated processes work. They create a graphical language for creating this representation. To quote:

> "A controller (automatic, human, or joint control) of a complex system must have a model of the general behavior of the controlled process....If an operator's mental model diverges from the actual state of the controlled process/automation suite, erroneous control commands based on that incorrect model can lead to an accident."

The authors also assert that one of the major factors that leads to an operator/machine mismatch is *lack of appropriate feedback*, especially when this feedback is needed to communicate *unintended side effects*. As such, they indicate when it is important for a semi-autonomous, complex device to communicate when its state has changed. It is important both that the human understand the current state of the device in terms that make sense to the human, and that these terms are somehow translatable into language the device understands.

2.5 Plug-and-Play Devices and Dialogue Systems

In [LRGB01] the authors discuss the development of a dialogue system for controlling devices in the home. A compelling feature of this system is that devices can be plugged in "on the fly" – their capabilities and a grammar for discussing these capabilities can be added dynamically to the dialogue manager. For instance, when a dimmable light switch is added to the system, an device model describing the dimmability of the switch is dynamically added to the system and a grammar that allows for utterances such as *Dim the light to fifty percent* is added. Three hierarchies exist in which the device must be placed:

1. The linguistic resources needed to query and control devices

2. The functionalities the devices implement
3. The code needed to control the devices

The goals of the system are exactly the same sort of goals pursued by the project described in this paper: plugability of devices into a dialogue system. However, while they have made a good start, they lack device models and descriptions for complex actions – while they can turn lights on and off, they wouldn't be able to control a mobile robot in a dynamic environment very easily, for example. The system they have built mostly deals with single utterances like "Turn on the light in the kitchen" which result in nearly instantaneous actions. In order to have more complex dialogues with more complex devices, actions that have duration must be considered, as well as actions which occur concurrently. Specifically, the relationship among these concurrently executing actions will be relevant to the dialogue at hand.

3 Project Outline

I have discussed above some of the research in several fields that is relevant to my project. There has been much fruitful work done on enabling dialogue systems to understand how humans communicate about plans they have formed. This work should serve as a basis for enabling intelligent devices to understand what it is the human operator wants it to do, and how it should be done. At the same time, while there has been much work in developing powerful control systems for autonomous devices, especially mobile robots, there remains much work to be done regarding how best to communicate about the actions of the devices which are carried out by devices so controlled. Indeed, there is a body of research that indicates that it is critical that human operators be able to understand the current "mode" of complex systems (here, the complex system in question is the intelligent device). Moreover, there has not been work on how to effectively convey a human user's beliefs about a plan to an intelligent device, in terms that it can understand; that is, there is a specific problem of converting from one rational agent's mental state representation of a plan to another's. Finally, there has been little investigation into how natural language can be used to make such communication more effective, such that *mode confusion* can be avoided, and the possibly many parameters involved in any one of a device's activities can be controlled simply and naturally.

This paper focuses on designing mechanisms for designing a class of dialogue systems for *task-oriented dialogues*, a specific subset of the range of "practical" dialogues discussed above. A *task-oriented* dialogue is one which is "focused on accomplishing a concrete task" – that is, it is a dialogue about accomplishing some specific task or tasks in a real or simulated environment. Allen, *et al*, hypothesize in [ABD+01] that general-purpose tools can be built for enabling dialogue management over the set of practical dialogues. Specifically, they formulate the *Domain-Independence Hypothesis*:

> "Within the genre of practical dialogue, the bulk of the complexity in the language interpretation and dialogue management is independent of the task being performed."

This claim both motivates the work described in this paper, and it is supported by the end result.

The two major goals of the project described here were derived from the exploitation of this hypothesis. Specifically, they are:

1. To create a scripting language similar in spirit to PRS-LITE which is powerful enough to control autonomous agents at a high level, but with special features making it particularly suitable for *communicating* in natural language about activities the device is currently engaged in, has completed, and should do in the future. Moreover, this language should support *joint-activities* which involve cooperation between the human operator and the intelligent device, an area often overlooked by robot programmers. This language should lead to a perspicuous run-time representation of the joint-activities in which the human and device are involved, such that this representation may be used as context to better understand and produce utterances related to the activities being performed.
2. To exploit features of natural language such that controlling intelligent devices is *easier* or *more natural* when they are dialogue enabled, as compared to controlling them with a GUI or a command line interface.

In addressing the first issue, I will introduce the notion of the *Activity Tree* as a means of tracking and modifying the status of multiple concurrent activities. In addition, I will discuss a particular representation of *Activities* I've developed. The representation allows the dialogue system to both talk about activities and understand utterances

that pertain to activities and their parameters. Moreover, it allows the dialogue manager to answer questions about *why* specific activities are currently being done or being planned. I will introduce a *recipe scripting language* that shares many features with PRS-LITE, but is designed to make communicating about activities straightforward.

In addressing the second issue of exploiting natural language, I will discuss many issues. My focus, however, will be a mechanism I've developed for specifying *constraints* over activities (for instance, "always fly high"). I will describe a system that allows the human operator to naturally specify and modify relatively complex constraints using natural language and then ensures that these constraints are adhered to. In addition, this system detects a wide-range of *implicatures* in order to ensure that the constraint set remains coherent, even when the user only *implicitly* removes constraints from the set and replaces them with new ones. It is my belief that such constraints are expressed easily in natural language, while they are difficult to communicate via a graphical user interface, and especially difficult for a naive user to express in a logic formalism. Moreover, I will show how the structure of the Activity Tree can be exploited by a system for managing constraints in order to allow for more flexible, natural, and robust dialogues.

Many of the examples cited in this paper derive from a dialogue system meant for controlling an autonomous helicopter. This is because the CSLI dialogue manager and the activity modeling/constraint management system presented here were made to work first in this domain. A toy version of the current system has been built for controlling an imaginary "robot butler," and previous, less-advanced, incantations of the system have been adapted for various other dialogue systems, including an in-car stereo controller and a voice interface to a scheduler, like the ones used on personal digital assistants (PDAs). The goal of the work presented here is to create a straightforward means of porting the dialogue front-end to further applications.

3.1 Fitting in a Dialogue System Architecture

In terms of existing architectures for dialogue systems, this project is meant to serve as a link between a dialogue manager and a device/agent. Because this framework represents information which is common across all task-oriented devices in a single format, the dialogue manager needs only to have algorithms which operate over this

more abstract structure. This saves us from having to make *ad-hoc* changes to the dialogue manager for each new device. Specifically, rather than building into the dialogue manager knowledge about each device for which it must faciliate dialogue, the goal is to provide the dialogue manager with general knowledge about how *task-oriented* dialogues work in general, and how *joint activities* are structured. The goal of this paper is to develop a framework by which such knowledge can be specified *declaratively*, so that general-purpose algorithms in the dialogue manager can operate over the declaratively defined information in order to facilitate complex *task-oriented* dialogues.

Throughout this paper, I will assume that a dialogue manager with a basic set of capabilities already exists. Specifically, I will assume that the dialogue manager is capable of doing the following:

- Converting natural language to *logical forms* which it uses internally as a semantic representation of natural language;
- Generating natural language from such logical forms;
- Keeping track of the *context* provided by previous discourse so that *dialogue games* like question-answer pairs and command-acknowledgement pairs can be produced and understood;
- Further using this context to do such things as determine the referent of *anaphoric expressions* and *aggregate* sets of utterances to be said by the system so that they flow conversationally (see *e.g.* [Ste01] on aggregation).

Moreover, I assume that this dialogue manager architecture can be interfaced, if desired, to the following components:

- An automatic speech recognizer,
- A parser,
- A graphical user interface,
- A text-to-speech synthesizer.

I make these assumptions because such a system exists in the form of the CSLI dialogue manager ([LGP02]). Moreover, similar systems have been built as was noted in section 2.1.2. The framework described in this paper is meant to further enhance such a dialogue system, though it is ambivalent, for the most part, with regard to the way in which the dialogue system actually provides most of this functionality. The dialogue-manager algorithms described in this paper have been

implemented as part of the CSLI dialogue manager, but they could also be used to extend other similar dialogue managers.

4 The Activity Tree and Activities

In order to dialogue enable task-oriented agents like mobile robots, it is critical that the dialogue manager maintain a representation of what the device it is controlling is actually doing, plans to do, and has already done at any given point in time. It is important that this representation, or at least some aspects of it, be convertible to natural language. That is, the dialogue system should be able to respond to questions of the form *What are you doing?*. Being able to answer questions like these is critical for avoiding operator confusion. The dialogue system should be able to describe the current state of the robot in a manner that the operator can understand. Moreover, a dialogue system should be able to answer, at at least some basic level, the question of *why* the device is behaving as it is at any given moment. In particular, it should be able to answer questions like *Why are you doing X?*. As was discussed above, the current control languages like PRS-LITE for complex devices often make the answering of such questions difficult because they don't produce an easily comprehensible picture of how the device operates.

In addition, since the goal is to allow task-oriented agents to *collaborate* with the human operator in *joint activities*, the representation should represent not just the tasks that the device is engaged in, but those that the human operator may be doing as well. The interactions between the two agents' activities should be understandable, so that the way in which particular actions by each agent give rise to coordinated joint actions is clear and describable in natural language. Moreover the dialogue system also needs to be able to communicate the desires and intentions of the human operator to the robot itself. As such, its representation of the state of the device should be rich enough so that the commands and corrections made by the operator can be accurately communicated to the system.

Given the above considerations, it is apparent that the representation we need is not one which includes every system-level detail of how the device (or human) will accomplish a given task. Rather, the level of detail should be appropriate to the actions which need to be *discussed* in order to accomplish the goals of the task. For instance, even

though we know generally that braking and accelerating are involved in the process of driving a car from one place to another, we might choose not to model this level of detail in the dialogue system. Perhaps for the dialogue system we are designing, we simply don't care about discussing the details of accelerating and braking – instead, we wish to focus only on a higher level of *granularity*. A dialogue designer, then, should be able to model the actions of the device and human at a level of granularity appropriate to the task at hand.

Even if we did wish to discuss fairly low level details about how the device works, our representation should not be committed to necessarily modeling how intelligent devices/agents *actually* perform actions. Rather, it should be geared toward the way in which humans *conceptualize* the actions being performed. That is, the representation should reflect the state of the device in such a way that a human operator can make sense of it. The representation should provide a *mapping* between the human's conceptualization of how a particular action is performed and the agent/device's, rather than modeling the way in which the robot actually does the actions necessary to achieve the goal. It should not be a model of the way in which individual circuits and motors of a robot work together to pick up a block, for example, but it should be decomposed in terms that the human operator can be expected to understand – perhaps, in this example, this would consist of bending the elbow, opening the fingers, closing them around the object, and then raising the arm. Just as humans do not generally speak at the level of firing neurons or stretching muscle fibers when they are giving instructions to one another, we should not seek to model this level of detail in a dialogue system. Indeed, it is exactly these sorts of details that we are attempting to get away from by using natural language to interact with devices: we would like to be able to interact with them on the level at which we conceptualize and understand the world, not the way in which they do. As such, it is crucial not that the representation in the dialogue manager be true to the inner workings of the device, but that it is able to represent these inner workings in a way which humans can readily conceptualize. Only in this case can the human operator successfully participate in joint activities with the device.

At first blush, it appears that simple slot-filling dialogues, like those discussed in section 2.1.1 above, might suffice for understanding the desires and intentions of the human operator – after all, a command and control system mainly needs to understand commands

from the human operator and then communicate them to the device. However, the dialogue snippets in (2) and (3) below demonstrate how *mode confusion* can easily arise if a dialogue system does not accurately model the sort of "common-sense" knowledge that a human operator knows about tasks that a device might perform. In other words, the dialogue system needs to understand not only what sort of activities the device and human operator are undertaking, but how these activities relate to larger goals and/or other activities.

(2) O: Patrol between the tower and the school
S: Okay. Now patrolling between the tower and Springfield school.
...
O: Fly to the tower at high speed.

The desire of the operator in (2) seems relatively clear. He or she desires that the helicopter patrol between the tower and the school, and moreover that when the helicopter is flying to the tower as part of this mission, it should do so at high speed. Another interpretation of the above dialogue is, however, that the operator at first wanted the helicopter to patrol, but then changed his mind and simply wanted it to fly to the tower at high speed. A dialogue that follows exactly this sort of pattern appears in (3):

(3) O: Patrol between the tower and the school.
S: Okay. Now patrolling between the tower and Springfield school.
...
O: Deliver the medical supplies to the tower at high speed.

The above dialogue follows the exact same sort of pattern as the other one, but (assuming the medical supplies are not currently onboard the helicopter, and they do indeed need to be picked up) its most felicitous interpretation is that the operator intends that the helicopter cancel (or at least suspend) its patrol operation and deliver the medical supplies immediately.

What's highlighted here is that the dialogue system needs to somehow be able to understand that the ways the operator's first and second utterances are related in (2) and (3) are different. In (2) the second utterance *elaborates* the first, while in (3) the second *contradicts*

(or *revises* or *stands in contrast to*) the first. The dialogue manager must be empowered with some common-sense knowledge in order to do this: specifically, it needs to know that "flying to the tower" is in fact a *sub-activity* of "patrolling between the tower and the school." It is not simply enough for the dialogue system to know that the helicopter is "patrolling" – but it also must know that a component of this "patrolling" activity is flying to a particular location.

In order to enable the dialogue manager to make inferences like the one above, as well as further inferences which I'll discuss later, I developed for this project the representation of an *Activity Tree* which can be embedded as part of the *Information State* of a dialogue manager. The Activity Tree is used as a medium of communication to and from the device and the dialogue manager; it is meant to represent the current state of the joint activities being undertaken by the operator and the device, or at least the portion of this state that is relevant for enabling meaningful discourse with the human operator. The Activity Tree consists of a tree of activities (see below), where the descendants of a particular activity are sub-activities of that activity. Each activity, at any given moment, must be in a particular *state*, where a valid state must be one of the following:

not_resolved : the activity has only been partially described by the user, not all of its parameters have yet been set

resolved : the activity has been fully described

request_send : the activity should be sent to the device to be planned, when it is appropriate to do so

planned : activity has been planned by the device planner

sent : the activity has been sent to the device (or operator) to execute

current : the device (or operator) is currently executing the activity

suspended : the activity has been indefinitely suspended

cancelled : the activity has been cancelled

done : the activity was successfully completed

skipped : the goals of the activity were already true, so it was skipped

failed_preconditions : the preconditions required for the activity to be executed do not hold

constraint_violation : the activity, as currently specified, violates one or more of the current constraints (see section 7)

conflicts : the activity has a resource conflict with other activities

The *Activity Tree* has a root, which will hold important information for the constraint management system which will be described later. The root, however, does not actually represent any particular activity the device (or operator) is, has been, or plans to be engaged in, and as such it has no state. Due to this lack of a true root, the Activity Tree is actually better viewed as a *forest* which contains trees of activities, where each tree may consist of currently executing or planned activities. In this way, the representation of the current state of the system mirrors that used by robot control languages like PRS-LITE, discussed in section 2.3. Such a representation is the natural result of having hierarchically structuring activities of which multiple ones may be executing concurrently.

The relatively simple Activity Tree for the patrolling example given above looks like this (at the moment in time when the helicopter has flown to the tower, then to the school, and is now flying back to the tower as part of its patrol operation):

```
root
..patrol_between (tower) (school) [current]
....go (tower) [done]
......take_off [done]
......fly_atom (tower) [done]
....go (school) [done]
......take_off [skipped]
......fly_atom (school) [done]
....go (tower) [current]
......take_off [skipped]
......fly_atom (tower) [current]
```

Note that nodes which are indented below other nodes are their *descendants*. For instance, `go (school)` is a child node of `patrol_between (tower)` in the above example. The state of each node is given in brackets – for instance, `[done]`. The conventions used in the above diagram will be used throughout this paper.

The Activity Tree is meant to be similar conceptually to the types of representations developed by Clark in [Cla96], and indeed it is meant to serve a similar purpose. Clark makes a powerful case in his book that an agent must be able to understand and model the joint activities in which he/she/it is engaged. Indeed, he shows that without such an understanding it would be impossible to comprehend

and produce utterances which both make sense given the current context, and further the accomplishment of mutual goals. The Activity Tree is meant to provide just such a context for the dialogue manager, so that it can enable the device to participate in meaningful and useful dialogues about the joint activities in which it is a participant.

5 The Recipe Scripting Language

While the *Activity Tree* represents the relationship among activities, the meshing of natural-language technology with agent-control technology becomes apparent when we examine the representation developed here to describe the activities themselves. Each activity on the Activity Tree is an instantiation of a *recipe* which comes from a *recipe library* for a particular device. Conceptually, this mirrors the proposals in [Pol90]; and it is similar in concept to the plan libraries in [ASF+95]. The recipes in the library, as well as particular properties of the library itself, are compiled from a *recipe script*, which must be written for the device that is being dialogue-enabled. The recipe script defines recipes for undertaking particular activities (often in the pursuit of particular goals). Each recipe models the domain-dependent common-sense knowledge which is needed to give rise to the structures on the Activity Tree which the dialogue manager uses for interpreting and producing relevant utterances. Moreover, each contains special constructs which are used by the dialogue manager to more effectively communicate, but which have nothing to do directly with the model of how a certain action is achieved (for example, the *Natural Language Mapping* and *Natural Language Slots* constructs which are described below).

The *recipe script* is formatted according to a special *recipe scripting language* designed as part of this project. The recipe script is designed so that recipes can be described in a powerful enough formalism to engage in joint activities with relatively intelligent agents or devices; at the same time, it requires constructs which make describing and querying about the activities instantiated from the recipes with natural language a straightforward task for the dialogue front-end to facilitate. Moreover, it is designed so that constraints (discussed in section 7) can be described that range over the contents of the instantiated activities. A recipe script consists of a *preamble* followed by a set of *recipes*, where each recipe can be *instantiated* as a particular

activity on a particular activity tree for a particular device.

Each recipe defined in the recipe script is added to the recipe library which is used by the dialogue manager to understand the capabilities of the device. A recipe can be conceptualized as consisting of the following components:

- A set of **slots**, similar in nature to the sets of slots used in form-filling dialogues, which represent the pieces of information needed before a recipe can be instantiated into an actual activity (or *plan*) capable of being executed by the device (or human operator).
- An **algorithm** (the recipe *body*) which operates over this set of slots that specifies how the activity should be decomposed further to accomplish its goals.
- **Device information** about the conditions under which the recipe may be executed (preconditions), the results of the actions described by the recipe (goals), the resources needed to perform the actions described by the recipe (resource list), and constraints over the way in which the actions will be performed.
- **Linguistic information** about how to describe under various circumstances (or when to refrain from describing) the instantiated activity as it is being performed.

In order to make the information in the recipe script available to the dialogue system, the script is first 'compiled' into a format that can be used more readily. This is done using a customized lexer-parser created in Java using ANTLR [Par00], a parser generator for Java. The following output files are generated by the compilation process:

1. `CSLI_ActivityProperties.java`: defines the activity properties
2. `myDevice.rep`, where `myDevice` is provided as part of the script: defines the body of each activity
3. `CSLI_TaskMatcher.java`: A simple class with a hash table to do NL mapping of command names
4. `domains.ecl`: Defines the domain of each slot
5. `constraints.ecl`: Provides the ECL^{i}PS^e predicates the Dialogue Manager will call

In this section and the next, I will discuss the components of the *preamble* and of the *recipe body*, why they are relevant to dialogue systems, and how the information is 'compiled into' the dialogue manager for use at runtime. For complete examples of scripts written in the formalism that will be described, please see Appendix B.

5.1 The Preamble

The preamble of the recipe scripting language has several sections which I will discuss here.

5.1.1 The Recipe Library

The first line of the preamble must be of the form:

`repfile "`*myRepFile*`"`

where `myRepFile` is the name of the file where the *recipe library* will be stored.

5.1.2 Type Definitions

Next, there must appear a `Types` section, in which valid slot types are defined. Defining a type, in this context, consists simply of defining the valid domain of values that slots of the given type may take on. These domains are used for reasoning about constraints (see section 7) – if slots of a particular type will never be involved in such reasoning, then their domains may be left unspecified.

Types are declared using the following format:
`Typename :: [` *value_1*`, ..., `*value_n*`];`
where each `value_i` is an allowable value for this type. If there are no such values, then the type does not participate in any constraint reasoning and it can take on *any value*.

An example of a Types section is the following:

```
Types {
  Speed          :: ["high", "medium", "low"];
  Altitude       :: ["high", "medium", "low"];
  Location       :: ["tower", "school",
                     "base", "lake"];
```

```
  MoveableObject :: ["water", "medical_kit"];
}
```

The above declares four types called *Speed*, *Altitude*, *Location*, and *MoveableObject* and assigns their respective domains.

5.1.3 Definable/Monitor Slot Definitions

The next two sections of the recipe script define all the valid *Definable Slots* and *Monitor Slots* that a particular recipe might have (please see the next section for a full definition of what exactly a definable slot is). These slots will represent the chunks of information the device will need in order to *instantiate* and *execute* the recipe. In particular, the *Type* of each slot must be specified, as well as each slot's minimum and maximum length. Additionally, the *default* value of the slot may be specified; for example, `medium` is specified below as the default for `toAltitude` and `toSpeed` Building upon the above type definitions, an example that comes from the WITAS domain is the following:

```
DefinableSlots {
  Location       toLocation:1-3;
  Altitude       toAltitude:1-3 = "medium";
  Speed          toSpeed:1-3 = "medium";
  MoveableObject carryObject:1;
}

MonitorSlots {
  Speed          curSpeed:1;
  Altitude       curAltitude:1;
  Speed          toAltitude:1;
  MoveableObject grippedObject:1;
}
```

While in the above examples there is a certain parallelism between the monitor slots and the definable slots, this is certainly not required by the formalism. The above definitions define definable slots named *toLocation*, *toAltitude*, and *toSpeed* each with a minimum length of 1 and a maximum length of 3 indices. The *carryObject* slot is defined as having only a single index (the helicopter can only carry one object at a time). The monitor slots are meant to be used to reflect the device's *state*; in the above example, slots named *curSpeed*, *curAltitude*, *toAltitude*, and *grippedObject* are defined, each with a single index. In the

WITAS domain, these slots are used to keep track of the helicopter's current state.

5.1.4 Resources

The final section of the preamble is the *resources* declaration section. Here, each of the *resources* that may be used by the various activities of the device must be declared. Each resource is simply a string. Here is an example from the WITAS domain:

```
Resources {
  uav;
  gripper;
  camera;
}
```

5.2 Components of a Recipe

After the preamble of the activity script appears a list of *recipes*, where each recipe can be instantiated into an activity. Each recipe consists of the following components:

1. An activity type. *e.g.*: `take_off`, `land_at_location`, `patrol_among_locations`
2. A ***N**atural **L**anguage Mapping* of the activity type
3. An *Agent Tag* indicating which agent should execute the activity
4. A set of *definable* slots which contain relevant parameters to the this activity, of which some may be *required* and others may be *optional*. *e.g.*: `toLocation` (the location to go to), or `toSpeed` the speed at which to fly there
5. A set of *monitor* slots which are meant to be filled at runtime with information about the *state* of the device. *e.g.*: *curLocation* (the current location of the helicopter), *curSpeed* (the current speed of the helicopter)
6. Resources. The set of resources that this activity needs. *e.g.*: A camera
7. Preconditions. A set of conditions which must be true in order to do this activity
8. Goals. A representation of the desired outcome of the activity
9. Banned. A set of "states of the devices" which are banned

10. **N**atural **L**anguage Slots. An association between an activity's state, and the detail to which it out to be described.
11. A Super Recipe. Each recipe may optionally inherit some of its properties from a *super* (or *parent*) recipe, in a fashion similar to other object-oriented, single-inheritance programming languages, like Java.
12. Body. A script which defines what this recipe does, when it becomes instantiated into an Activity.

Below, I will describe each component of a recipe, the syntax for defining it, and the manner by which the information it embodies is 'compiled' into the dialogue manager by the "recipe compiler."

5.2.1 Activity Type, NL Mapping, and Agent Tag

Each recipe is given a particular *Activity Type*, which is simply a unique name for the recipe. The NLMapping of a recipe describes the verb that should actually be output by the system. So, while a designer might end up assigning a particular recipe an Activity Type of *patrol_between_search*, this activity can have an NL Mapping of "patrol" – which is the verb that will be used when the system actually discusses this activity. The NL mapping exists because several distinct (though often related) concepts in a language may be mapped to a single verb – for example, while the concepts invoked by "patrol between the tower and the school" and "patrol at the tower for a blue car" are distinct (and in the formalism provided here, this distinction is captured by having two distinct recipes), both concepts are captured in English by the single verb *patrol*.

Each recipe must also declare an *agent tag* which identifies the agent who should execute activities which instantiate the particular recipe. In the current system, only the tags `USER` and `SYSTEM` are supported: where `USER` refers to actions that the human operator should take, and `SYSTEM` refers to the device being controlled. In theory, this set of tags could be expanded to include more types of agents and might be changed to allow for a list of agents, all of which could potentially complete the action.

The activity type, NL Mapping, and agent tag are defined at the beginning of the definition of each recipe. The syntax is as follows:

`taskdef<`*activity_type*`, "`*nl_mapping*`"> `*agent_tag*`{`

```
//rest of recipe definition goes here
}
```

During the recipe compilation process, the mappings from activity type to natural language mapping are written to a hash table which can be accessed by the dialogue manager through methods provided in `CSLI_TaskMatcher.java`. The agent tag is stored as part of the recipe in the recipe library.

5.2.2 Definable Slots

The set of *definable* slots are those slots whose values must be specified before an activity can be executed by the device. For instance, in the WITAS project, before the helicopter can fly somewhere, it must know to *where* it should fly. Hence the activity *go* contains a definable slot named *toLocation*, which is meant to hold the location to which the helicopter should fly. Typically, definable slots correspond roughly to the arguments of a verb, (here, *fly to the* ***tower***), or potentially to other modifiers like adverbs (*fly* ***quickly***) – though there is nothing in the formalism which actually requires this.

The set of *required* definable slots are those slots which *must be specified explicitly by the operator, or inferred directly from an operator's (possibly multi-modal) utterance.* On the other hand, the set of *optional* slots are those slots whose values can be be filled in by default values, or through *constraints* (to be discussed later). The syntax for declaring a required slot is the following:

```
required Type SlotName;
```

While an optional slot is declared like this:

```
optional Type SlotName;
```

Consider, as an example, the recipe for *transporting* an object from one location to another. In the WITAS system, this activity contains the following set of definable slots (which are requisite as indicated and are meant to correspond to the concepts noted):

- *fromLocation* (required: the location from which to pick up the object)
- *toLocation* (required: the location at which to drop the object)

- *carryObject* (required: the object to carry)
- *fromSpeed* (optional: the speed at which to fly to the first location)
- *fromAltitude* (optional: the altitude at which to fly to the first location)
- *toSpeed* (optional: the speed at which to fly to the second location)
- *toAltitude* (optional: the altitude at which to fly to the second location)

Using the types declared in section 5.1.2, the above slots could be defined as part of a recipe as follows:

```
DefinableSlots {
  required fromLocation;
  required toLocation;
  required carryObject;
  optional fromSpeed;
  optional fromAltitude;
  optional toSpeed;
  optional toAltitude;
}
```

Note too that the above assignments of *required* and *optional* make sense because it is imperative that the system know what object to pick up, from where to pick it up, and where to drop it. The speed and altitude it should fly at, while necessary parameters, are not critical, in some sense, to the activity. While the activity can be successfully accomplished no matter what their values are, the activity cannot even, in some sense, be *defined* unless the required slots are filled with values. The required definable slots, then, make up the *core* notion of the activity.

The *required* flag, then, is a means by which the dialogue manager knows when it should initiate a *slot-filling dialogue*. If the operator specifies only some of the required slots in his or her initial command, then the dialogue manager will ask information-seeking questions of the operator until all of the *required* slots are filled. On the other hand, *optional* slots can be filled in with *default* values if they are not explicitly mentioned or filled in through constraints – they do not

merit a slot-filling sub-dialogue initiated by the dialogue manager. This will be further discussed in much greater detail in section 7.

Each slot also has associated with it a particular *type* defined in the *preamble* (see section 5.1.2). In the WITAS system, for instance, *fromLocation* and *toLocation* are of type *Location*, while *carryObject* is of type *MoveableObject*. The effect of assigning a type to a particular slot is that the *domain* of the slot becomes limited: *carryObject*, for example, can only be assigned to an object that can, indeed, be carried by the helicopter (the domain associated with a particular type is defined in the *types* preamble of the recipe script).

The set of definable slots of a particular instantiated recipe (aka *activity*) is accessible at runtime to the dialogue manager. In section 7, I will discuss how a slot's type (and hence, its domain) is relevant for constraint reasoning.

5.2.3 Monitor Slots

The monitor slots are simply a way to reflect information about the current state of the device at any given time. Defining which slots are important to a recipe simply makes it more efficient to calculate the constraints (to be discussed later) over the recipe. Like definable slots, the monitor slots of a particular activity are accessible to the dialogue manager at run time through the `CSLI_ActivityProperties` class. The syntax for declaring monitor slots is identical to that for definable slots, except no `optional` or `required` prefix is used, since this concept does not apply to monitor slots.

For example, the monitor slots for the `go` recipe in the WITAS domain are as follows:

```
MonitorSlots {
  curLocation;
  curAltitude;
  curSpeed;
}
```

5.2.4 Resources

It is often important to understand what *resources* are needed in order to complete a particular activity. For instance, in the WITAS domain, it is important to understand that flying somewhere requires using the entire helicopter, while taking a picture of something merely

requires use of an on-board camera. This information can be used by the dialogue manager to detect resource conflicts and initiate intelligent dialogues about them. For instance, the CSLI dialogue manager generates the following dialogue in the WITAS domain:

(4) O: Fly to the school.
S: Now flying to the school at medium speed and medium altitude.
...
O: Deliver the medical supplies to base.
S: Delivering the medical supplies to base conflicts with flying to the school.
Should I deliver the medical supplies to the base now or later?
O: Now please.
S: Okay.
I have suspended flying to the school.
Now flying to the hospital. [In order to pick up the medical supplies]
...

The resource conflict is detected by the dialogue manager and a relevant sub-dialogue is initiated. In order to facilitate such dialogue, each recipe includes a section in which the resources required by any activity which instantiated this recipe are listed according to the following syntax:

```
Resources {
  resource_1;
  resource_2;
  ...
  resource_n;
}
```

During the recipe compilation, this information is stored as part of the recipe in the recipe library.

5.2.5 Preconditions

The preconditions are the set of conditions which must be true before an activity that instantiates a particular recipe can be executed. These conditions are expressed in terms of predicates over the monitor and definable slots. These too are simply stored as a list which is part of the recipe in the recipe library. The list is not simply a string, but

a list of `CSLI_Expression` objects which are designed for the efficient manipulation of first order logic expressions.

5.2.6 Goals

These are the goals that an instantiation of a recipe is meant to achieve. These too are expressed in terms of predicates over the values of the monitor and definable slots. For instance, to express that the goal of the "fly_to" activity in the WITAS system is to actually move the helicopter to a particular location, we might write the following goal: *curLocation == toLocation* meaning that the location we were meant to fly to should be equal to the location where we actually are. The goals as well are simply stored as a list of `CSLI_Expression` objects in the recipe library entry for a particular recipe.

5.2.7 Banned

Each recipe may contain a list of logical relationships among the values of slots which may be banned. For instance, to express the constraint that the helicopter shouldn't drop objects while at high altitude, we write as part of the recipe for the activity type "drop" the following item on the banned list: *curAltitude == "high"*. As I will discuss later, these constraints are defeasible. That is, rather than defining impossible states of the world, they define states of the device that under normal circumstances should be avoided. The banned list is compiled into a similar format as the goals list during recipe compilation.

5.2.8 NLSlots

When the state of an activity changes, the dialogue manager often reports this state change. For instance, when an activity becomes `current`, the dialogue manager will give a report like *Now flying low to the tower at high speed.* As a result, the dialogue manager must be able to map from an activity to a natural language (or multi-modal) representation of that activity. In this process, the filled-in slots of the activity are considered, as well as the activity type, in order to produce a meaningful utterance. As activities become more complicated, with more and more slots, it becomes unwieldy for the dialogue manager to talk about all the parameters of an activity in each utterance about that activity. For instance, when an activity has completed, it's not always necessary or desirable to convey an entire description to the

user. When the helicopter has flown to its destination in the WITAS system for example, (5a) is preferred to (5b).

(5) a. I have flown to Springfield school.

b. I have flown low to Springfield school at high speed.

Since the goal of the activity was to arrive at the location, this is the information that is really relevant. How the helicopter got there is not so important, especially considering that this information has already been negotiated by the operator and the system has already announced its intention to fly to the school at low altitude and high speed.

In pursuit of these ideas, the recipe scripting language includes the ability to associate the reporting of activities in particular states to particular slots. For instance, to specify that for the activity of flying we'd like the helicopter to report its destination, height, and speed when it's actually doing that activity, but not when it's reporting the completion of the activity, we write the following lines of script:

```
NLSlots {
  current: toLocation, toSpeed, toAltitude;
  done: toLocation;
}
```

This indicates to the dialogue manager that it should report the destination (toLocation), target speed, and target altitude of the helicopter when it announces that it is currently flying somewhere, but only the destination when it has reached somewhere. The results are dialogues similar to the following:

(6) O: Fly low to the school at high speed.
S: Now flying low to the school at high speed. [System reports: `toAltitude`, `toLocation`, `toSpeed`]
...
S: I have flown there. [System reports: `toLocation`]

In addition, the script also allows for the left hand side to be `default` – which gives the default set of slots to report in all states not explicitly mentioned.

As activities have more and more slots, and hence become more complex and unwieldy to talk about (for instance, fighting a fire, or transporting objects), the ability to have easy control over generation becomes extremely useful. Indeed, for some activities, you might wish

to have "hidden" slots that the system should never try to discuss – or only discuss in rare circumstances.

The set of NL slots for each state, as well as the default set (if supplied), are stored as part of the particular recipe in the recipe library.

5.2.9 Super Recipes: *extends* and *abstract*

Each recipe may optionally *extend* other recipes. In the context of the recipe scripting language, this simply means that the recipe will inherit the values of all the sections listed above, with the exception of the *body*, *nlmapping*, and *activity type*. Moreover, certain recipes may be declared as *abstract*, meaning that they are not meant to actually ever be instantiated into activities but only that they should serve as super recipes to other recipes. For example, in the WITAS domain, there is defined an abstract recipe called *move*, which embodies the concept of moving, but can't actually be instantiated. Instead, there is another recipe called `go`, which should actually be instantiated when the helicopter is instructed to fly somewhere. The description of these two activities is given in figure 1.

5.2.10 Body

Each recipe optionally contains a body, which is a body of code written in a specialized activity scripting language which I discuss in section 6. If a recipe doesn't contain a body, then it is assumed to be *atomic* – that is, if it becomes instantiated then it should be sent to the device to actually be executed, rather than further decomposed.

6 The Recipe Body

The body of a recipe consists of a script which defines a *recipe for action* – in the sense discussed in section 2.2 – that describes the actions which ought to be performed in order to accomplish the recipe's goals. In the formalism described here, we can think of the recipe body as the *algorithm* which describes what should be done in an abstract sense, and the *slots* as the data over which the algorithm operates. While the recipe body for the activity of *fighting a fire* describes abstractly what's involved in fighting a fire (continually picking up water from a particular location, carrying that water to the location where the fire

Figure 1: move and go – Inheritance and Abstract Recipes

```
abstract taskdef<move,"move"> {
  DefinableSlots {
    required toLocation;
    optional fromLocation;
    optional toSpeed;
  }
  MonitorSlots {
    curLocation;
    curSpeed;
  }
  Resources {
    uav;
  }
  Banned {
    toSpeed == "zero";
  }
}

taskdef<go,"go"> extends move {
  //inherits locations,speed from move
  DefinableSlots {
    optional toAltitude;
  }
  MonitorSlots {
    curAltitude;
  }
  Banned {
    toAltitude == "zero";
  }
  NLSlots {
    default: toLocation;
    current: toLocation, toAltitude, toSpeed;
  }
  //definition of Body -- removed for this example
}
```

is, and then dropping it until the fire is out), it is not until the *definable slots* for this recipe are filled in that it can be *instantiated* into an activity which can be performed, since these provide the requisite information such as where to pick up the water and where the water ought to be dropped. By requiring the algorithm described by the recipe body to operate over the values of the slots, a direct connection is made between the linguistic aspects involved in the task-oriented dialogue and the tasks at hand which are being performed.

The script which composes the recipe body was conceived in order to balance two objectives. First, it is meant to characterize joint-activities (to be done by the human and the device together), as well as activities carried out only by the device, in a way that matches the way humans conceptualize doing activities. That is, it should match the way that humans actually think about and understand recipes for joint activities rather than simply represent the way in which the device actually carries out a specific action (as was discussed in section 4). On the other hand, it must be compatible with the representation utilized by the intelligent agent or device to the degree that the device can actually execute atomic actions specified in the script for it to do. If it meets both of these objectives – that is, if it is both compatible with the way that humans conceptualize activities and capable of decomposing into terms that the device can work with – then it can successfully act as an intermediary between the human and the device. When the device performs the actions described by the script, then the human will be able to understand *why* these actions are being done. On the other hand, when the human operator seeks to modify the way in which a particular action should be done, the device will be able to understand these desires in terms of the data/parameters (*i.e.* the slots) over which a recipe operates. Moreover, because slots are also linguistically motivated, the recipes should be easy both to describe and to understand descriptions of using natural language.

Toward the end of matching up with the representations needed by an intelligent device or agent, the script which makes up the recipe body is designed to be extremely similar to the scripts that are actually used to control mobile robots. It is based loosely on the ACT formalism, which is now included as part of the COLBERT [Kon97] scripting language in Saphira, a software package distributed by ActivMedia Robotics with its mobile robots. While, at first, actually using one of these languages seemed tempting, I chose instead to write my own script interpreter and language to better pursue the goal of

making the script match up with the way that humans conceptualize activities. First, I wanted to naturally and explicitly be able to make references to the slots defined for the activity. Second, I wanted to try to balance the language so that it would be simple enough, and straightforward enough, that natural language could be generated to describe it. COLBERT can be interfaced directly to C, and I worried that the expressiveness of a full fledged programming language like C would be difficult to talk about using natural language and, more importantly, I wanted the language to provide a framework which would be conducive to designing recipes in a way which would make them match up with the way humans conceptualize activities – as the relationship among concurrent activities generated by COLBERT can often be difficut for a human to understand. Moreover, I worked under the premise that the sorts of plans that users would want to talk about (in terms of how they were further decomposed) would not be arbitrarily complex. That is, they would be the sorts of plans that a person could describe in a few sentences. In particular, I worried about the complex interaction between global variables that can sometimes be found in mobile robots actually using COLBERT and Saphira. Much communication among activities which are running simultaneously is often done through the setting of global variables – such behavior makes the relationship between different activities extremely difficult to describe and their interaction abstruse. Rather than write a scripting language appropriate for writing all activities that a mobile robot could ever do, my goal was to write a scripting language that described the sorts of activities that a person might reasonably be expected to describe and want to talk about. In particular, I assumed that the operator wouldn't want to be concerned with the fine details involved in complex robot actions – for example, keeping the robot localized as it moves to a location, or ensuring that its pitch and yaw are correct if it is a flying robot. Rather, I assumed that the user would want to discuss activities *at the level at which he or she might discuss the activities of the robot if he or she were planning joint action with another person* – a level at which such details as how movement is accomplished and headings are maintained are not discussed.

The scripting language, then, consists of the following commands inspired in large part by the ACT formalism (brackets indicate optional parameters):

- `intend activity(slot_assignments) [blocking] [act_name]`

- `stop act_name`

And the following loop constructs:

- `repeat {...}`
- `do {...} while(conditions)`
- `foreach(assignments) {...}`

6.1 *intend* and *stop*

Intend and Stop are commands to create new activities and stop ones which are running. Activities which are *intended* are sent to be planned and executed.

6.1.1 *intend*

The process of intending an activity is one of attempting to load the recipe with the name activity and instantiating its slots with the values given in `slot_assignments`. The `slot_assignments` link a particular slot in the spawning activity with the particular slot in the child activity. For instance, consider the simple body of the activity for "go" in the WITAS system. It is as follows:

```
Body {
   intend take_off(toAltitude=THIS.toAltitude);
   foreach toLocation t, toAltitude a, toSpeed s {
       intend fly_atom(toLocation=t,toAltitude=a, toSpeed=s);
   }
}
```

This decomposes `go` into a `take_off` and a series of `fly_atom` activities. For the `take_off` activities, the toAltitude slot of the `take_off` activity is linked to the `take_off` slot of the `go` activity. Similarly, the `toLocation`, `toAltitude`, and `toSpeed` slots of the `fly_atom` activities which will be spawned, are linked to particular indices of their corresponding slots in the parent activity of "go." This is done using the *foreach()* loop construction, which I'll discuss later.

It is important to note that the slot assignments between parents and children are not like those made in many traditional programming languages. The value of the parent activity's slot is not simply copied over into the child's slot upon creation of the task. Rather, the two actually (for all intents and purposes) *share* the slot. It is as if

the value of the slot were passed in *by reference*. Hence, if the value of `toLocation[0]` changes in the `go` activity, then the corresponding `toLocation[0]` will be updated in the `fly_atom`. The slots are structure shared.

To understand why such structure sharing is important, consider the following example dialogue:

(7) A: Patrol between the tower and the school.
B: Okay, now flying to the tower at medium altitude and medium speed.
A: Fly there at high altitude.
B: Okay.
B: Now flying to the tower at high altitude and medium speed.
...
B: I have flown there.
B: Now flying to the school at medium speed and medium altitude.
...
B: I have flown there.
B: Now flying to the tower at high altitude and medium speed.

Note that each successive act of flying to the tower is done at high altitude, not just the particular instance of it which was modified by the user. This occurs because the first index of the `toAltitude` slot of patrol is shared with its child `go`. When its child's slot+index is modified by the operator, then, so too is its slot+index. Hence, when it spawns the second instance of flying to the tower, the operator's instruction is retained appropriately.

6.1.2 *stop* and *noblock*

In the above example, all of the spawned activities were (by default) *blocking*. That is, until the `take_off` activity was completed by the system, it did not try to fly anywhere. Mobile robots, however, often have several activities running *simultaneously*. In order to support this, I followed the conventions of the COLBERT programming language, and allowed for the ability to spawn child activities in a non-blocking fashion, as in the following snippet from the `find` activity in WITAS:

```
Body {
```

```
  intend locate(searchItem = THIS.searchItem);
  intend track(followItem = THIS.noticedItem) myTrack noblock;
  intend identify(searchItem = THIS.searchItem);

  stop myTrack;
}
```

In this example, the system first tries to locate an object in the world (for example, a red car) via the `locate` method. When this activity is done (and hence, an object matching the desired description has been located), the recipe executor then spawns a new child task of `track` – which essentially follows the car and keeps it in sight. The *noblock* keyword is used to indicate that the executor should go ahead and continue executing the code that follows, even before the tracking activity is completed. In addition, this particular instance of track is assigned a name – *myTrack* – so that it can be referenced later. In particular, it is passed to the *stop* command, which halts the activity after the object has been identified.

6.2 Loops: *repeat*, *do...while*, and *foreach*

The looping constructs behave as in most procedural languages. A *repeat* loop simply repeats the contents inside of its braces endlessly, until the activity is explicitly stopped by the *stop* command, or cancelled by the user. The `do...while` loop executes its contents forever, or until the *condition* of the while(*condition*) becomes true. This condition can be direct equality and inequality statements over the values of particular slots, or calls to predicates over these values. In order to test the more complex predicates (for example, in the WITAS system, there is a predicate for `fire_out` which tests if a particular fire is out in the world) at runtime, the execution system defines an interface that the domain-specific predicates must implement in order to have their values tested, as will be discussed later.

The *foreach* construct is a specialized version of the more standard "foreach" constructs in programming languages that allow for the iteration of a list. In this context, a foreach loop iterates over all the *filled-in* indices of a slot, *or a set of slots*. By allowing for the simultaneous iteration over more than one slot simultaneously, I allow for more subtle relationships between slots. For instance, in the code from `go` above, each `fly_atom` activity is instantiated with a parallel

set of `toLocation`, `toAltitude`, and `toSpeed` parameters. This allows for `go` to decompose flying to several sequential destinations at several different altitudes and speeds in a convenient manner – that is, the slots can be used as parallel arrays. This ability has been used extensively in recipes created for the CSLI dialogue system.

7 Constraints and Defaults

As devices become capable of more complex behavior and the number of parameters that can be set for each activity grows, it becomes desirable to express *constraints* over the values of those parameters. For instance, in the WITAS system, the operator should be able to control many parameters that dictate *how* the helicopter should fly, such as *speed* and *altitude*. At the same time, it would be unwieldy if the operator were required to specify these parameters each time he or she gave the helicopter a new command – for instance, to fly to a specific location. In pursuit of this, my activity model supports three relevant notions: *optional definable slots*, *defaults*, and *constraints*. As was discussed above, *optional definable slots* are those slots whose values need not necessarily be assigned explicitly (or inferred directly) from the operator's commands or answers to questions posed by the system. Instead, such slots may take on their values through the use of *defaults* and *constraints*.

The motivation for *defaults* and *constraints* emerges out of every day observations about the way that people use language to describe activities. Most common activities involve some set of parameters which don't necessarily need to be specified – for instance a person can walk or drive quickly or he can talk or sing loudly, but he can also simply walk or drive, talk or sing. When I ask a person to walk from point A to point B, it doesn't matter so much how fast he walks, but just that he actually succeeds in walking between point A and point B; nonetheless, he will still have to perform the action at some particular speed. On the other hand, I can specifically assert that while walking between point A and point B, he should walk at a speed of 2 miles per hour. Then, if he were to walk at 5 miles per hour between A and B, we would say that he had not done the action which I commanded him to do. Here, then, we see that *speed* for the activity of *walking* is optional, and were we to create a recipe script for walking, *speed* would be an *optional* slot. Moreover, it could be assigned some default value,

say 3 miles per hour, which would be used when speed wasn't explicitly specified – since walking must be done at *some* specific speed.

From a more practical point of view, defaults become useful in dialogues with possibly complex devices simply because if they don't exist, dialogues can become tedious. Consider, for example, the contrast between the dialogues where no defaults exist in (8a) and (8b) and one in which defaults are used in (8c).

(8) a. O: Fly to the school.
S: Okay. At what speed should I fly?
O: Medium speed.
S: Okay. At what altitude should I fly?
O: Medium altitude.
S: Okay. Now flying to the school and medium altitude and medium speed.

b. O: Fly to the school at medium altitude and medium speed.
S: Okay. Now flying to the school at medium altitude and medium speed.

c. O: Fly to the school.
S: Okay. Now flying to the school at medium altitude and medium speed.

Once activities exist which have default values that are filled in automatically, as in (8c), it becomes immediately desirable that there should be a straightforward means to redefine these defaults on the fly. Perhaps a deadline is approaching, and I need you to help me with several tasks – making copies of a presentation, delivering the copies, and sending off several letters. Rather than telling you in turn to do each task quickly, I might simply say something like "Please do everything I ask you to do today quickly." In essence, I have, at least temporarily, redefined the default speed at which I'd like you to do all the actions I ask you to do. Moreover, I've defined a *constraint* which identifies how you should do all actions I've asked you to do, and *all future activities which haven't even yet been specified.* That is, rather than just changing the parameters of specific actions I've asked you to do, I've issued more general guidelines which also apply to future activities as well. For example, consider the contrast between the dialogue in (9a) in which defaults can't be redefined and the one in (9b) where they can.

(9) a. O: Fly to the school at high speed and high altitude.
S: Now flying to the school at high speed and high altitude.
...
O: Fly to the tower at high speed and high altitude.
S: Now flying to the tower at high speed and high altitude.

b. O: Always fly high and fast.
S: Okay.
...
O: Fly to the school.
S: Now flying to the school at high speed and high altitude.
...
O: Fly to the tower.
S: Now flying to the tower at high speed and high altitude.

Once we have the power to redefine defaults in a natural way, it becomes immediately clear that such utterances seem to belong to a larger class of dialogue moves, which I'll call *constraint specifications*. For instance, it seems just as natural to negate the values that certain slots can take on, or perhaps even to specify more complex constraints such as disjunctions. Consider the sample utterances in (10) which I claim also belong to this natural class of dialogue moves.

(10) a. O: Never fly high.

b. O: Always/Never fly low or fast.

The dialogue moves in (8c), (9b), and (10) then seem to make up a natural class that could be fruitfully used across a wide-range of devices and agents. Moreover, the class appears to be natural in the sense that humans often take these sorts of dialogues for granted because they have underlying assumptions about the importance of various "parameters," their default values, and the way that the values that can fill in these parameters can be constrained. While robot designers or programmers may be used to thinking about the various parameters that a robot program or function might take, people generally make implicit assumptions about the default values of parameters of activities, assumptions which only become salient when other pressures arise (like an upcoming deadline). Moreover, people already have natural ways of expressing constraints in natural language, while it is more difficult and complex (as will be discussed below) to express the interaction between these constraints in a formalism which an intelligent device can handle. The conversational intelligence implicit in

understanding the interaction between constraints and defaults, however, can be applied across a wide-range of task-oriented dialogues; as such it is an ideal candidate for modularization. In this section, I will discuss how the framework presented here models defaults and constraints by building on the recipe/activity representations already discussed; moreover, I will show how the model can be used to facilitate dialogues like those in (8c), (9b), and (10), as well as others which will be motivated later.

7.1 Defaults

This section very briefly introduces a basic algorithm for processing defaults; this algorithm will be revised to take into account the interaction between constraints and defaults in section 7.4.1. In the recipe script, each definable slot can be assigned a default value in the slot declaration section (see section 5.1.3). For slots related to speed, for instance, the WITAS system assigns a default value of `"medium"`. If all of the *required definable slots* have been set, then any optional slots which have not been explicitly assigned a value will be assigned their appropriate default values before the activity is sent to the device to be executed.

7.2 Constraints

While the ability to have default values is useful, it is not sufficient to totally free the operator from dialogues such as that in (9a). Put in terms of the representation of activities/recipes developed so far in this paper, this dialogue was frustrating because in it, the operator was forced to be constantly specifying lists of values to be assigned to optional slots for which the default value was not the desired value. Such difficulties, as well as the benefits that arise from understanding utterances such as those in (10), motivate a generic interface for specifying and managing *constraints* over slot values – an interface which is presented here.

At the simplest level, these constraints allow a means for the operator to redefine defaults, with utterances such as *Always fly high.* Constraints can be much more powerful than this, however. They can also include negations, as in *Never fly high*, conditionals such as *When flying to the school, fly low*, or disjunctions such as *Always fly*

at low altitude or at high speed.[1] In principle, the system is capable of handling arbitrary first-order-logic formulas involving the values of the slots of activities – however, what subset is actually within the range of human competence is an open question which the following discussion will hopefully shed some light on (though I don't propose to actually supply a precise answer to the question). In future work, this would certainly be an interesting question to tackle.

Constraints are implemented here as formulas in first order logic over the values of slots (potentially over both monitor slots and definable slots – though I've focused mainly in definable slots in my development). The actual translation from an utterance of a constraint to its first order logic representation (and the reverse: the generation of an utterance describing a constraint based on its formula) is the responsibility of the dialogue manager. However, as I will discuss later in the section on dialogue management, I have provided tools to make a large subset of these translations relatively straightforward – and as domain-independent as possible. In this section, I will simply assume that first-order-logic constraints come in from the user via a "black box" and that utterances pertaining to them can be mapped directly from them.[2]

Constraints are associated with particular activities on the activity tree. In particular, each activity holds two lists of constraints: a *banned* list and a *necessary* list. Formulas that appear on the banned list are those which should evaluate to false when the activity is executed. For instance, if the operator were to instruct the system *Don't fly high*, then the formula corresponding to *fly high* would be added to the banned list of a particular activity. Conversely, the *necessary* list contains those formulas which must evaluate to true when the activity is instantiated. While these lists could logically be combined into a single list, in order to better manage the dialogue and more easily express the constraints in terms understandable to the operator, they are separated into the two lists depending on how they were specified by the operator.

There are two types of constraints: *global* constraints and *local* constraints. Global constraints apply to all current and future activities

[1] Disjunctive utterances are not currently supported by the current Dialogue manager, though the constraint management system described here would have no problem handling them.

[2] In the CSLI dialogue system, this "black box" is the Gemini parser/generator [DGA+93] which makes use of a grammar developed at CSLI

– for example, the constraint *Always patrol at high altitude* is global in the sense that it should be "applied" to all current and future instances of patrol tasks. On the other hand, local constraints are those which apply only to a particular activity. For example, if the user were to first command the helicopter to patrol at the school, and then tell it to *Don't do it at low altitude* or *Do it at low altitude or low speed* – then this constraint should be applied only to the particular instance of the patrol activity in question.

In order to support this distinction between global and local constraints, the Activity Tree implements a system by which it "trickles down" the *banned* and *necessary* constraint lists. This trickling down simply has the effect that each activity, in addition to being subject to the constraints on its own *banned* and *necessary* lists, is also subject to all of the constraints of its *ancestors*. Moreover, the root of the tree is a special activity which has no slots, but does contain *banned* and *necessary* constraint lists. The dialogue manager, then, assigns global constraints by adding them to the constraint lists on this root node. When new activities are instantiated, they then inherit all of the root node's constraints via the trickling down mechanism, and hence are subject to global constraints.

For example, consider the Activity Tree below. Here, `N` is the set of *necessary* constraints at a given node, `CN` is the *complete* set of *necessary* constraints at a given node, including those constraints "trickled down" from above. Similarly, `B` is the set of *banned* constraints at a given node, while `CB` is the *complete* set of *banned* constraints at a given node, including "trickled down" constraints from above.

```
root N={n1,n2}, CN={n1,n2}, B={b1}, CB={b1}
..act1 N={n3}, CN={n1,n2,n3}, B={}, CB={b1}
....act2 N={}, CN={n1,n2,n3},B={b2},CB={b1,b2}
..act3 N={n4}, CN={n1,n2,n4}, B={b3}, CB={b1,b3}
```

In addition to the *banned* and *necessary* lists, each activity also has two corresponding lists: the *ignoreBanned* and *ignoreNecessary* lists. These lists contain constraints that should NOT be inherited from ancestor nodes in the tree. These lists allow for specific activities to ignore global constraints (or constraints expressed over other ancestor activities), if the operator instructs that this should be the case (as in the dialogue in (11)).

Adding these lists (notated `IN` and `IB` for *ignoreBanned* and *ignoreNecessary* respectively) to the example above, yields:

```
root N={n1,n2}, IN={}, CN={n1,n2}, B={b1}, IB={}, CB={b1}
..act1 N={n3},IN={n1},CN={n2,n3},IB={},B={}, CB={b1}
....act2 N={},IN={},CN={n2,n3},IB={b1}, B={b2},CB={b2}
..act3 N={n4},IN={n2}, CN={n1,n4}, B={b3}, IB={b1}, CB={b3}
```

An example in which a constraint would be added to the *ignoreNecessary* list is the following:

(11) O: Always fly high.
...
O: Fly low to the school please.
S: Just a minute...I am supposed to always fly high, should I fly low to the school anyway?
O: Yes.
S: Okay. [Add "fly high" to *ignoreNecessary* list]

In this case, the constraint that the helicopter should always fly at high altitude is relaxed, but only in the context of a specific activity. If the operator were to later command the helicopter to fly somewhere else, the global constraint of always flying high would still be in effect. This is because the activity tree looks something like this:

```
root N={"fly high"}, IN={}, CN={"fly high"}, ...
..go (school) N={}, IN={"fly high"}, CN={}, ...
```

Hence, future descendents of the `root` will still be subject to the `"fly high"` constraint, however children of the `go` activity will not longer have this constraint "trickled down" to them. This is a critical distinction, because if the constraint were simply removed from the constraints list at the root, then it would no longer apply to future activities.

7.3 Examples of constraints

Constraints are implemented as first order logic statements over the values of particular slots – or more specifically, over the values of the specific indices of specific slots. Consider the formulas in figure 2 and their appearance on either the banned or necessary lists of the *root* of the activity tree (all are actually supported by the current dialogue front end, except where noted):

Figure 2: Constraints translated to formulas on the *banned* and *necessary* lists

necessary:

- *always fly high*: [command="go" → toAltitude="high"]
- *always fly at low speed*: [command="go" → toSpeed="low"]
- *when patrolling at springfield school, patrol at low altitude*: [(command="patrol" ∧ toLocation="s1") → toAltitude="low"]
- *always fly low or fast*: [command="go" → (toAltitude="low" ∨ toSpeed="fast")] [a]

banned:

- *never fly high*: [command="go" ∧ toAltitude="high"]
- *never fly at low speed*: [command="go" ∧ toSpeed="low"]
- *never patrol at springfield school at low altitude*: [command="patrol" ∧ toLocation="s1" ∧ toSpeed="low"]
- *never fly low and fast*: [command="go" ∧ toAltitude="low" ∧ toSpeed="fast"]

[a] Not supported by the current dialogue manager

From figure 2, it is apparent that the sorts of constraints which can be represented are far more powerful than the simple redefinition of "defaults" discussed in section 7.2. Indeed, in principle a constraint can be an arbitrary first-order-logic formula over slots, or slots and particular indices. This allows for a relatively wide range of constraints. The main difficulty is in converting from natural language to coherent formulas and back, but below it will be shown that this can be done in a relatively domain-independent form for a large number of interesting cases.

It is interesting to note that the constraints on the *necessary* list appear in the form of conditionals in which the command (or *activity type*) is always part of the antecedent. It is critical to note that it would be incorrect to simply have a conjunction of slot assignments appear on the necessary list. For instance, placing the formula `[command="patrol"` $\wedge$ `toAltitude="high"]` on the *necessary* list would require that *all activities be of type patrol*, which is clearly incorrect. Indeed, an important part of translating constraints from natural language to FOL is determining which slots should appear in the antecedent and the consequent respectively. It is here that the distinction between *required* and *optional* slots plays another important role: when translating constraints of the form "always *command*" (where *command* is a type command that might be given to the system), the slots which are *required* should appear in the antecedent while those which are *optional* should appear in the consequence. Indeed, this definition is merely the formalization of the more vague distinction I initially presented – that *required* slots are those which form the *core concept* of an activity, while *optional slots* are those which define possible parameters of an activity, but which do not compose part of its core concept.

7.4 Constraining Activities and Interfacing to ECL^{i}PS^e

There are several problems that need to be solved in the course of implementing a system of constraints that can be utilized by the dialogue manager to engage in relevant dialogue. This project solves some of these problems, but some remain to be dealt with in future work. First and foremost is the problem of taking a (possibly) partially specified set of slots for an activity and then applying some appropriate combination of defaults and constraints to fill in the remaining slots. It

is when this operation fails that the two other major problems arise. The first of these arises when the partial specification itself is inconsistent with the current constraint set. In this case, the system needs to determine which set of constraints has been violated and report this to the operator. The second of these occurs when a default value can't be assigned to a slot because it would violate a constraint, but the constraints are not specific enough to require a single particular value for the slot. Essentially, the combination of the constraints and the partial specification of an activity do not uniquely define a single activity.

For example, consider the following three sample dialogues below (with dialogue moves noted in brackets):

(12) O: Always fly high. [specify global constraint]
...
O: Fly to the school please. [command]
S: Now flying high to the school at medium speed. [report]

(13) O: Always fly high. [specify global constraint]
...
O: Fly low to the school please. [command]
S: Just a minute...
I am supposed to always fly high, [report constraint violation]
should I fly low to the school anyway? [relax constraint y/n question]
O: Yes. [y/n answer]
S: Okay. [confirmation]

(14) O: Never fly at medium altitude. [specify global constraint]
...
O: Fly to the school please. [command]
S: What altitude should I fly at? [slot value query]
O: Low altitude. [slot value answer]
S: Okay.
...
S: Now flying low to the school at medium speed.

In order to facilitate these dialogues, the dialogue system must be capable of carrying out the following algorithm:

Algorithm: CONSTRAINT DIALOGUE GAMES

1. Calculate the number of possible activities that are defined by the combination of the partially specified activity, the constraints over that activity, and the default slot values which can be assigned without violating constraints. Let this number be w.
2. IF $w = 1$ THEN success has been achieved and an activity has been uniquely identified (as in (12))
3. IF $w = 0$ THEN determine the most informative set of constraints has been violated and report it to the operator (as in (13))
4. IF $w > 1$ THEN determine which slot(s) are underspecified and spawn a relevant information-seeking dialogue (as in (14))

In order to supply the dialogue manager with the necessary information, I made use of ECL^{i}PS^e ([WNS97], [ACD$^+$02]), a constraint-based solver that extends Prolog. In order to interface ECL^{i}PS^e to the existing CSLI Java-based infrastructure, ECL^{i}PS^e was run as an embedded process within the CSLI Dialogue Manager (see [NSSS02] for technical details of how this is accomplished).

ECL^{i}PS^e solves constraint satisfaction problems by taking the following steps:

First, each variable must be assigned a particular *domain*. The domain of a variable can be an integer or real number range, or a particular set of atomic values.. For example, to set the domain of the variable X to be $\{high, medium, low\}$ the following construct is used: $X :: [high, medium, low]$
Next, constraints are declaratively defined in terms of the values of the variables. For example, to constrain X such that it can only take on the set of values $\{high, low\}$, we declare the following constraint:

```
(X #= high) #∨ (X #= low)
```

where `#=`, for example, indicates the assignment predicate. Finally, we ask ECL^{i}PS^e to produce all sets of *labellings* of variables assigned to values, such that the constraints are satisfied.

The constraint management system, then, makes use of ECL^{i}PS^e

by assigning each index of each slot a particular variable, issuing constraints over these variables, and then asking ECL^{i}PS^e to return the set of all possible *labellings* such that the constraints are satisified. As mentioned above, each index of each slot is assigned a unique String which identifies the name of the variable which will be used in ECL^{i}PS^e to constrain that particular slot+index. The first step, then, is to set the domain of each of these variables. The domain of each variable is originally set by the system designer as part of the recipe script: recall that each slot is associated with a particular *type*, and that each *type* is assigned a particular domain when it is declared. During the recipe "compilation" process, each slot's domain is determined and a bit of ECL^{i}PS^e code that defines a predicate called *set_domain* is generated in a file called *domains.ecl* which assigns each "eclipse variable" that corresponds to a slot+index to the domain of the *type* associated with the slot. This predicate is loaded at runtime, and called as the first step in the constraint-satisfaction process.

Next, each constraint from the relevant activity's *banned* and *necessary* lists (as well as those constraints inherited from ancestors, but not on the ignored list) must be translated into the appropriate ECL^{i}PS^e constraints. Recall that each constraint is expressed as a FOL expression over the values of slots – either over the value of *all* indices of a particular slot, or over the value of a *particular* index of a particular slot. In the case where a constraint is over a particular slot+index, the translation into ECL^{i}PS^e is straightforward. The appropriate variable name that corresponds to that slot+index is identified, and the constraint is output in terms of that variable. However, when a constraint is meant to apply across *all* indices in a particular slot, the process is not as simple.

In this case, the most straightforward approach would be to simply apply the constraint across all indices of the slot. This proves problematic, however. Consider, for example, the basic fly activity in the WITAS system – called *go*. This activity contains the slot *toLocation*, which actually has three indices (which means that the activity can be used to fly to three locations in sequence). It is a common occurence, however, for only the first of these indices to be specified; for instance, if the operator gives the command "fly to the hospital" then only the first index of *toLocation* will be assigned a value. In this case, we don't want to force the uninstantiated second and third indices to be assigned values, since the *minimum* length in the recipe script required of *toLocation* is set to be 1. In pursuit of this, we assign the

special value 'null' to the variables that correspond to the second and third indices of *toLocation*. In order to allow this assignment, 'null' is included as a possible value in the domain of every variable – for variables that we actually want to have assigned a value, we stipulate the additional constraint that that variable's value cannot be equal to 'null' when calling ECL^{i}PS^e. The basic assumption, then, is that a constraint applies over all the indices of a slot up to its minimum length, and those beyond its minimum length which are actually filled in.

This basic assumption, however, must be modified when we examine a scenario in which the operator has commanded the system to "fly to the tower then the school." In this case, the above algorithm works fine for assigning constraints to the slot *toLocation*, however it runs into problems when we attempt to assign constraints to the values of indices of *toSpeed* and *toAltitude*. Both of these slots are meant to be "parallel" to *toLocation* in the sense that the values at corresponding indices in the three slots are passed together to the atomic activity `fly_atom`. In this example, the second value of *toLocation* is `"school"`, and whatever values are assigned to the second index of *toSpeed* and *toAltitude* respectively will control the way in which the helicopter flies to the school, but not the tower.[3] As such, we need to indicate both the first and second indices of *toSpeed* and *toAltitude* should be subject to constraints and assigned values, even though by our basic assumption above we would only end up requiring that the first index of both *toSpeed* and *toAltitude* be assigned values, since the minimum length of each slot is 1 and all indices of each slot are unassigned.

Since this sort of parallelism is *domain dependent*, the constraint management system defines a callback method *getSlotMaxLengthForConstraint()* which takes as parameters the name of the slot and the instance of *CSLI_ActivityProperties* which that slot comes from. This callback method is a part of *CSLI_ActivityBase* and its default implementation is to follow the basic assumption given above. Super classes of *CSLI_ActivityBase* should define this method if the default implementation is insufficient. In the case of the WITAS system, this method is overridden such that when the slot in question is *toSpeed* or *toAltitude*, the maximum number of indices to be constrained is

[3]While this system of parallel slots may seem overly complex at first, it arises from the need to be able to interpret commands like "fly to the tower at low speed and to the school at high speed"

calculated based on the content of *toLocation*. This callback method allows arbitrarily complex relationships to hold between different slots while at the same time it frees the constraint management system from having to understand these constraints.[4]

With this callback method in hand, the sytem now has a means of translating from constraints over slots and their values to a meaningful representation in ECL^{i}PS^e. Constraints are assigned to the first n indices of the slot, where n is determined by calling *getSlotMaxLength-ForConstraint()*. For constraints which are over multiple slots, this expansion must occur recursively. Finally, if the constraint comes from the *banned* list, then it must be negated – since ECL^{i}PS^e supports only constraints which necessarily must be adhered to. In the following example, I will show the results of this process on a few constraints, given the specified partially instantiated *CSLI_ActivityProperties* and assuming that the relevant ECL^{i}PS^e variables assigned to the zeroeth index of *toLocation* is `ToLocation0`, and similarly for other slot+index pairs:

Given Activity Properties with the following slots specified ("fly to the tower then the school"):

```
command = go
toLocation[0] = tower
toLocation[1] = school
```

The *necessary* constraint "always fly high" $command = go \rightarrow toAltitude = high$ yields the set of constraints:

```
Command#=go #--> ToAltitude0#=high
Command#=go #--> ToAltitude1#=high
```

The *banned* constraint "never fly high" $command = go \wedge toAltitude = high$ yields the set of constraints:

[4] An interesting bit of future work would be to make some of this knowledge *declarative* and include it in the recipe script. For instance, parallel slots in the sense defined in this paper could be declared as such and this knowledge could then be automatically integrated into the dialogue manager. As we will see later, this knowledge also plays a key role in the Noun Phrase resolution procedures which need some domain knowledge about the relationships between slots in order to work properly

```
¬(Command#=go #∧ ToAltitude0#=high)
¬(Command#=go #∧ ToAltitude1#=high)
```

Finally, the partially specified activity is convered into constraints as well. This is a straightforward process: all values that are assigned in the activity are converted to simple equality constraints of the form: `SlotIndexVar #= value`, where $SlotIndexVar$ is the appropriate variable that corresponds to the slot+index in question, and value is that value that has been assigned to that slot+index. At this point, we can simply query ECL^{i}PS^e for the set of all suitable sets of variable assignments that satisfy the constraints.

7.4.1 Dealing with Defaults

In order to actually determine how a partially specified activity should be properly instantiated, there is another aspect of the problem to consider: defaults. Each slot which is not linguistically specified by the operator (or inferred directly from the operator's commands), may have a suitable *default* value. As discussed above, default values for slots are declared as part of the recipe script. The simplest way of dealing with defaults would be to simply assign them appropriately to all slots that have not been already assigned a value, and then run the resulting activity specification through ECL^{i}PS^e to determine if it meets the constraints set out by the operator. This approach, however, is clearly unsatisfactory. Consider the following dialogue that might emerge from such an algorithm, assuming that the default speed at which to fly is set to `medium`:

(15) O: Always fly at high speed
S: Okay
...
O: Fly to the tower
S: Just a minute ...
I am supposed to always fly at high speed
Should I fly to the tower at medium altitude and medium speed anyway?

As the above dialogue illustrates, if we were to assign defaults to all unspecified slots BEFORE calling ECL^{i}PS^e, then the constraints set out by the user can't be used to fill in unspecified slot values to which they pertain – in this case, the *toSpeed* slot.

To solve this problem, the following algorithm is used:

Algorithm: CONSTRAIN INSTANTIATIONS
Given: a partially specified activity, P.

1. Consider the finite set of slots+indices which can potentially be assigned a default value given a particular partial activity specification; call this set of slot+index to value assignments S.

2. Consider each subset $s \subseteq S$ in order from largest to smallest, assign the slot+index to value assignments in s to P, yielding P'.

   (a) Send P' along with the *necessary* and *banned* constraints to ECL^{i}PS^e to yield W: the set of all legal assignments of values to variables in P'.

   (b) If $|W| = 1$, succeed and return W.

   (c) If $|W| > 1$, retain W and continue iterating. If future iterations do not succeed, return W.

   (d) If $|W| = 0$, continue iterating. If this is the last iteration, and there are no previously retained Ws, then fail.

This algorithm tries to find the largest number of defaults that can be assigned to yield a single legal (subject to the constraint set) fully-instantiated activity (all definable slots filled with a value). It prefers, however, to find exactly one fully specified activity in a set to assigning as many defaults as possible. If, after not assigning any of the defaults, it still can't find any possible instantiations, then it fails because the partially specified activity itself must violate the constraint set. If exactly one legal instantiation is found at any point, then the algorithm succeeds immediatly and returns this result. If there are always more than one possible instantiations, then the algorithm prefers the set that arises from instantiating as many default values as possible.

If the algorithm succeeds in finding a single possible instantiaton, then the dialogue manager accepts this instantiation and goes ahead and requests that the activity be executed. If there are zero possible instantiations, then the dialogue manager reports that the partially specified activity violates the constraint set. Moreover, the constraint

management system allows the dialogue manager to request which set of constraints were violated, so that it can inform the operator (more about this in the next section). If there are multiple possible instantiations, then the dialogue manager engages the user in an information seeking dialogue – specifically, it determines which slot+indices cannot be assigned a unique value based on the constraints, and asks the operator to supply values for these slots. An example of such a dialogue appears in (14).

Note, that just as we used the method *getSlotMaxLengthForConstraint()* above to deal with the case of so-called *parallel slots*, a similar mechanism must be used for defaults as well. Consider that if two locations are specified to which the helicopter should fly (in the WITAS system), then we should consider the first two indices of the slots of `toLocation` and `toSpeed` when assigning defaults. In pursuit of this, a callback method called *getSlotMinLengthForDefault()* is defined which returns the minimum number of indices in a given slot which ought to be assigned default values (if possible).

7.4.2 Determining which set of constraints has been violated

In the case where the partial activity specification supplied by the operator is determined to violate the current constraint set, the constraint management system provides a means for the dialogue manager to determine specifically which subset of the constraint set was violated. This allows for dialogues like the one in (13).

Of course, it is simple to find one easy solution to the question of which set of constraints was violated: *all of them.* If we were to remove all constraints, then the partially specified activity would not violate any constraints at all! The constraint management system aims to find the *most informative* subset of constraints that were violated; in this context, this means the *smallest* subset of constraints. While it is technically true to say that if we removed both the constraints "always fly high" and "always fly at high speed" then the command "fly to the tower at high speed and low altitude" would become legal – but this is not nearly as useful as determining that only the constraint "always fly high" is the one that is actually causing the problem. As such, the system tries to find the smallest subset of constraints which need to be removed in order for the partial activity specification to be allowable (it does not, however, prefer one subset of constraints over

another if both sets are the same size – in this case, it merely reports the subset that it finds first).

In order to determine which set of constraints was violated, we simply iterate over all possible subsets of the constraint set and find the largest one which doesn't prohibit the partially specified activity. The algorithm is as follows:

Algorithm: FIND VIOLATED CONSTRAINTS
Given: a partially specified activity, P
Given: the set of all *banned* and *necessary* constraints, C

For $i = 1$ to $|C|$ {

For each subset $c \subseteq C$ such that $|c| = i$ {

1. Let $Q = C - c$
2. Send P along with the constraint set Q to ECL^{i}PS^e to yield W: the set of all legal assignments of values to variables in P.
3. If $|P| = 0$ return c

}

}

With the set of violated constraints in hand, the dialogue manager can produce dialogues like the one in (13).

7.5 Maintaining a Consistent Set of Constraints

When humans use natural language to specify and understand constraints, they are not always explicit about certain underlying assumptions that they make. For instance, consider the following two sections of two different dialogues:

(16) a. O: Always fly high.
S: Okay.
...
O: Always fly at low speed.
S: Okay.

b. O: Always fly high.
S: Okay.

...
O: Always fly low.
S: Okay.

At the syntactic and semantic levels, there is nothing to distinguish (16a) from (16b); however, they are actually quite different. In (16a) there is an implicit "and" between each constraint specification – that is, the sequence of utterances is meant to specify a *conjunction* of constraints. By specifying the second constraint, the operator meant to add an additional constraint. In constrast, in (16b), the operator first specified one constraint and then implicitly *changed* this constraint later. Hence, in this case, at the end of the dialogue there should only be a single constraint in effect, namely: "always fly low." We might say that there is an *implicature* which must be calculated by the utterance of the second constraint specification in (16b) which doesn't exist in (16a), namely that the previous constraint specification should be cancelled.

If such implicatures are not understood by the dialogue system, then the set of constraints it maintains is in danger of becoming *inconsistent* in a sense that will be explored here. Continuing the example in (16b), if the dialogue manager were to fail to simply add both constraints specified by the user to the *necessary* list, then the list would contain both of the following constraints:

(17) 1. `command="go"` $\rightarrow$ `toAltitude="low"`
2. `command="go"` $\rightarrow$ `toAltitude="high"`

If this set of constraints were passed to an ECL^{i}PS^e in order to instantiate a partially instantiated activity like "fly to the school," then ECL^{i}PS^e would be unable to fully instantiate the activity due to the inconsistancy in the constraint set.

Despite this difficulty, these two constraints are not *logically inconsistent* in the sense that a formula such as $A \wedge \neg A$ is. While $A \wedge \neg A$ describes a situation which cannot be satisified in any possible world, the constraints in (17) are not so prohibitive; for example, we could always simply choose an activity besides `go` – we might take a picture of a car, for example. As such, the property of consistency we are after for the constraint set is not that of logical consistency, but something I'll call *commonsense task consistency* in this discussion. The challenge for the constraint management system, then, is to identify when the second utterance in constraint command pairs like the ones

in (16) gives rise to an implicature, so that *commonsense task consistency* in the set of constraints can be maintained. In the context of the constraint management system developed here, this means that when new constraints are specified, the existing constraint set must be searched for constraints which should be removed.[5] For the case in which constraints are truly arbitrary first-order-logic expressions, this is an incredibly difficult problem however. However, as we have seen, the range of constraints people are actually likely to give (which can actually be translated to FOL) is constrained such that only certain patterns are likely to emerge. Given this more limited problem, certain patterns which give rise to implicatures can be identified and then the constraint management system can look for these patterns to maintain a consistant constraint set. For the CSLI dialogue manager, the set of implicature patterns which needed to be recognized by the dialogue manager in order to keep the constraint set consistent (such that it only contained constraints which could actually be dealt with by the system) was identified – they are given in figure 3. In these patterns, the first formula corresponds to the first constraint-specification utterance in a dialogue and the second corresponds to the second. Note that we take advantage of the fact that items on the *banned* list in the form $A \wedge B$ can be rewritten as necessary constraints of the form $A \rightarrow \neg B$. When this rewrite is done, the assumption is made that the *definable* slots should appear on the left, and the *optional* slots should appear on the right hand side.

The example patterns in figure 3 illustrates cases where the constraint in the first line is replaced by the constraint in the second line. These patterns are limited to cases in which only 1 or 2 conjuncts are given. Of course, in a real system, these formulas should be generalized to instances in which 3 or more conjuncts appear – and, indeed, in the CSLI system such a generalization has been made. Where a single conjuct occurs in contrast to two earlier conjuncts (as in, for example, the pair: $[A = a \wedge B = b] \rightarrow C = c_1$ and $A = a \rightarrow C = c_2$), the more general case is one in which the second set of conjuncts are a *subset* of the first (here, $\{A\} \subseteq \{A, B\}$).

In the current constraint management system, all *necessary* constraints are broken down into their simplest form before being added to the *necessary* list. That is, a potentially complex constraint like *fly to the tower at high speed and at high altitude* is converted into two

[5]Or at the very least, which the dialogue system ought to bring up in some clarification subdialogue aimed at determining what the operator really meant

Figure 3: Implicature Patterns

1. $(A = a \wedge B = b) \rightarrow C = c_1$
   e.g. Always fly to the tower at high altitude.
   $(A = a \wedge B = b) \rightarrow C \neq c_1$
   e.g. Never fly to the tower at high altitude.
2. $(A = a \wedge B = b) \rightarrow C = c_1$
   e.g. Always fly to the tower at high altitude.
   $(A = a \wedge B = b) \rightarrow C = c_2$
   e.g. Always fly to the tower at low altitude.
3. $(A = a \wedge B = b) \rightarrow C = c_1$
   e.g. Always fly to the tower at high altitude.
   $A = a \rightarrow C \neq c_1$
   e.g. Never fly at high altitude.
4. $(A = a \wedge B = b) \rightarrow C = c_1$
   e.g. Always fly to the tower at high altitude.
   $A = a \rightarrow C = c_2$
   e.g. Always fly at low altitude.
5. $(A = a \wedge B = b) \rightarrow C \neq c_1$
   e.g. Never fly to the tower at high altitude.
   $(A = a \wedge B = b) \rightarrow C = c_1$
   e.g. Always fly to the tower at high altitude.
6. $(A = a \wedge B = b) \rightarrow (C = c_1 \wedge D = d_1)$
   e.g. Always fly to the tower at high altitude and high speed.
   $(A = a \wedge B = b) \rightarrow C \neq c_1$
   e.g. Never fly to the tower at high altitude. [6]
7. $(A = a \wedge B = b) \rightarrow (C = c_1 \wedge D = d_1)$
   e.g. Always fly to the tower at high altitude and high speed.
   $(A = a \wedge B = b) \rightarrow (C \neq c_1 \wedge D \neq d_1)$
   e.g. Never fly to the tower at high altitude and high speed.
8. $(A = a \wedge B = b) \rightarrow (C \neq c_1 \wedge D \neq d_1)$
   e.g. Never fly to the tower at high altitude and high speed.
   $(A = a \wedge B = b) \rightarrow (C = c_1 \wedge D = d_1)$
   e.g. Always fly to the tower at high altitude and high speed.

constraints representing *fly to the tower at high speed* and *fly to the tower at high altitude* respectively. More formally, given the constraint given in (18), the two constraints in (19a) and (19b) are actually added to the *necessary list*.

(18) `command= "go"` $\rightarrow$ `(toSpeed = "high"` $\wedge$ `toAltitude = "high")`

(19) a. `command = "go"` $\rightarrow$ `toSpeed = "high"`

b. `command = "go"` $\rightarrow$ `toAltitude = "high"`

As such, the algorithm in figure 4 has been implemented to identify constraints in the current set which should be removed given a new constraint specified by the operator. This algorithm makes use of the following simple helper functions:

- $isNec(c)$ which simply returns *true* iff c is on the *necessary* list (or is supposed to be added to the necessary list)
- $diff\_assign(r_1, r_2)$ returns *true* iff r_2 contains an assignment to a slot given in r_1, but the value assigned to that slot is different. For example, if r_1 was $\{A = a_1\}$ and r_2 was $\{A = a_2, ...\}$ then the function would return true.
- $same\_assign(r_1, r_2)$ returns *true* iff r_2 contains an assignment to a slot given in r_1, and the value assigned to that slot is the same. For example, if r_1 was $\{A = a_1\}$ and r_2 was $\{A = a_1, ...\}$ then the function would return true.

8 Algorithms for the Dialogue Manager

I will not discuss here the full details of the CSLI dialogue manager; I refer the interested reader to [LGP02]. Instead, I will discuss the interface provided by the activity modeling and constraints/defaults system and the services that any dialogue manager wishing to be enabled with it must provide. Fuerthmore, I will discuss algorithms which have been implemented in the CSLI dialogue manager which take advantage of the framework discussed in this paper in order to faciliate more natural dialogues between the human operator and the device. The system is built in such a way that there is nothing that ties it by necessity to the details of the CSLI dialogue manager; it does not rely on any one particular theory of discourse representation,

Figure 4: Algorithm for detecting implicatures

Algorithm: FIND CONFLICTING CONSTRAINTS DUE TO IMPLICATURE
Given: a new constraint, c_n to be added to the constraint set
Given: the set of all *banned* and *necessary* constraints, C, converted to canonical form[a]

Let I be an initially empty set to hold the constraints C which are found to be inconsistent with c_n.
For each $c \in C$ {
  Let r_c be the set of equality statements on the right side of c
  Let l_c be the set of equality statements on the left side of c
  Let r_{c_n} be the set of equality statements on the right side of c_n
  Let l_{c_n} be the set of equality statements on the left side of c_n

  If $l_{c_n} \subseteq l_c$ {
    If(
    $(isNec(c) \wedge isNec(c_n) \wedge l_{c_n} \subseteq l_c \wedge diff\_assign(r_{c_n}, r_c)) \vee$
    $(isNec(c) \wedge \neg isNec(c_n) \wedge |r_{c_n}| = 1 \wedge same\_assign(r_{c_n}, r_c)) \vee$
    $(\neg isNec(c) \wedge isNec(c_n) \wedge r_{c_n} \subseteq r_c)$
) {
      Add c to I, if $c \notin I$

    }
  }

  If($isNec(c) \wedge \neg isNec(c_n)$ {
    Let C_a be the set of $c_o \in C$ such that l_c = l_{c_o}
    Let r_{C_a} be $\bigcup r_{c_o}$
    If($r_{c_n} \subseteq r_{C_a}$) {
      Add c to I, if $c \notin I$
    }
  }

}

[a]Where *canonical form* corresponds to the form of constraints given in figure 3. That is, all necessary constraints are of the form $(A = a \wedge \ldots \wedge B = b) \rightarrow (C = c \wedge \ldots \wedge D = d)$ and all banned constraints are in the form $(A = a \wedge \ldots \wedge B = b) \rightarrow \neg(C = c \wedge \ldots \wedge D = d)$

any specific set of algorithms for processing or producing discourse, or any particular parser or grammar. That said, it is up to the dialogue manager to provide these services.

The system provides several capabilities that a dialogue system should interface to, including: a means of representing the relationship between different activities that are currently being executed, have been executed, or which are planned; the application of constraints and the ability to determine which constraints are problematic; and a representation of which slots are pertinent to an activity in various states. It also depends on the dialogue manager for several abilities, including: interpreting commands and constraints in natural language; the generation of reports about an activity based on its slots and their values as well as the generation of constraints in natural language; and knowledge about which slots may be "parallel." In this section, I will discuss each of these items and how they have been handled by the CSLI dialogue system.[7]

8.1 Translating Commands from Natural Language into Activity Representations

I've defined activities in this paper in terms of an *activity type* and a set of *slots* some of which are filled in with particular values. I've said nothing about how to translate commands like *Fight the fire at the tower* into these representations, or how to generate from them reports like *Now flying low to the tower at high speed.* In the CSLI dialogue manager, user utterances are parsed and system utterances are generated using a bi-directional unification grammar written using SRI's Gemini system [DGA+93]. The grammar for the CSLI dialogue manager has been hand-designed to be used for the command and control of mobile robots. I will not discuss the grammar here in detail, but suffice it to say that the dialogue manager deals only in the *logical forms* that are produced by the grammar and never with the actual surface string.

The logical forms for commands and reports generally break them down into their verb and the verb's argument PPs and NPs, as would

[7]I am indebted to Oliver Lemon, Laura Hiatt, Randolph Gullett, and Elizabeth Bratt for the hard work they have put into many areas of the dialogue system, including many of those elements that were necessary for interfacing to the activity modeling and constraint services I've discussed in this paper. Much of the work discussed in this section on the dialogue manager side of things was actually implemented by them.

be expected. For example, the logical form for *deliver the medical supplies to the school* is:

```
[command([deliver],
         [param_list([arg([np([det([def],the),
                               [n(medical_supplies,pl)]
                              ])]),
                      [pp_loc(to,arg([np([det([def],the),
                                          [n(school,sg)]])]))]])
          ])]
```

Similarly, the logical form for *Fly to the tower and the school at high speed* is:

```
[command([go],
         [param_list([[pp_loc(to,arg(conj,
                                     [np([det([def],the),
                                          [n(tower,sg)]])],
                                     [np([det([def],the),
                                          [n(school,sg)]])]))],
                      (speed),value(high)])])]
```

Given logical forms like these, the Dialogue Manager must go through two steps to translate it into a activity description. First, it must pull apart the arguments into the appropriate slots for the activity type. Second, it must "resolve" the NPs to determine what actual objects in the world they refer to.

In the CSLI dialogue manager, the CSLI_Activity class supplied by the device interface is subclassed by CSLI_Task so that extra information can be added on to each activity. Of interest here is that corresponding to any definable slot which can be described in terms of an NP (for instance, *toLocation* might correspond to the NP "springfield school"), CSLI_Task defines a slot to hold the NP associated with that slot – for instance, *toLocationNP*. The dialogue manager first pulls out the NPs for each command that belong in a slot and puts them in the corresponding NP slot. It also pulls slot values which aren't parsed as NPs, but rather as "mods" – modifiers such as "at high speed" – and puts them directly in their corresponding slots, for instance *toSpeed*.

Next, each NP is "resolved" through either dialogue context of database lookups to an actual entity in the world. If no such entity

can be found or multiple possible matches are found, then the dialogue manager initiates information seeking dialogues with questions like *Which tower do you mean?* and *Where is the pond?*. Once the NPs are resolved, the IDs of their referents are placed in the correct slots – for example, the id of the referent of *carryObjectNP* is placed in *carryObject*. Recall that the fact that it is the identifier of the referent that must be stored in a particular slot when determined in the recipe script when that slot was determined a type, which had a particular domain. Hence, what sort of value is actually stored in each slot can be determined on a domain-specific basis.

If some of the required slots still need values at this point, then the dialogue manager presents information-seeking questions, like *Where should I fly to?* to the operator. Once all of the required slots have been filled in, the activity's state is set to `resolved`, at which point the constraint and defaults management system kicks in and tries to fully instantiate the activity according to the current constraint set.

8.2 Translating Constraints from Natural Language into Logic Expressions

Currently, the CSLI dialogue system can understand constraints such as:

- Always/Never fly at high speed.
- Always/Never patrol at the tower at low altitude.
- Always/Never fly high and fast.

Essentially, the system can understand any normal *command* input prefixed by either *always* or *never*. This is because the algorithm it uses to translate these constraints into logic expressions is based on the algorithm above for translating commands in natural language to activity/slot representations. For example, *always patrol at the tower at high altitude* is simply interpreted by passing the *patrol at the tower at high altitude* through the command parser and making note with a flag that it was a global constraint. Then, the slots which are filled in are turned into a constraint where all of the *required* slots appear on the left hand side of the implication and the *optional slots* on the right hand side, as discussed above. I note again, here, the importance of the distinction between required and optional slots.

8.3 Translating activities and constraints into natural language

The process of going from an activity to natural language basically involves going through each slot and choosing an appropriate noun phrase, prepositional phrase, or other modifier to represent that slot. Each slot is marked in the dialogue manager either as being one of a PP slot, an NP slot, or a modifier slot in the context of particular activity types. An appropriate phrase is chosen by the dialogue system based on the contents of the slot, and then all of the phrases are assembled to create an appropriate logical form. As I mentioned above, in the recipe script different slots can be associated with each activity state. Hence, depending on the state of the activity, or the intent of the utterance (is it to be used for a question or a report, for instance), different set of slots may be used to generate the natural language to describe the activity.

To translate a constraint into natural language, it is first converted into an activity representation with the appropriate slots filled in which are covered by the constraint. Then, this is converted to a logical form and embedded inside the appropriate logical form for the various sorts of constraints – for instance, it is embedded inside a different form depending on whether it came from the *banned* or *necessary* list and whether it is a global or local constraint.

8.4 Avoiding Mode Confusion

A major problem facing a dialogue system for controlling devices is the problem of deciding *when* changes in the state of the world should be described to the user. In order to avoid *mode confusion*, it is imperative that the operator's beliefs about the state of the world sufficiently match the actual state of the device. For instance, if the device has successfully completed an activity that it was pursuing, then the operator needs to be informed of this so that he or she maintains an accurate mental model of what the device is doing. If the device does not keep the operator abreast of its state, then incomprehensible dialogues like the following might ensue:

(20) O: Fly to the school and look for a red car.
S: Now flying to the school
...and looking for a red car
Helicopter finishes flying to the school but says nothing

O: Cancel flying to the school

The above is just a small example of the sorts of problems that may occur if the operator does not maintain a consistent picture of the state of the world that matches reality and the mental state of the device. In this section, I discuss the techniques developed as part of the CSLI dialogue manager for avoiding *mode confusion*; in particular, I focus on how these methods are facilitated by the representation afforded by Activities, the Activity Tree, and Constraint Management System.

8.4.1 Announcing State Changes

The Dialogue System must somehow decide when to make announcements about when the state of the world has changed. The Activity Tree provides one mechanism by which the dialogue system can make intelligent decisions about what changes in the world are conversationally appropriate, and which need not be mentioned. Whenever the *state* of an activity on the tree changes (for instance from `planned` to `current`), the dialogue manager is notified of this update and can choose whether or not to announce this change in state. The simplest strategy is to announce every state change of every activity; however, this leads to some odd dialogue sequences. Consider, for instance, the a simple case from the WITAS system in which the helicopter flies from base to the school. Just before reaching the school, the activity tree looks like this:

```
root
..go (to school) [current]
....take_off [done]
....fly_atom (to school) [current]
```

Upon reaching the school, first the `fly_atom` activity becomes *done* and then, in turn, the `go` activity becomes *done* as well. Yielding first this activity tree:

```
root
..go (to school) [current]
....take_off [done]
....fly_atom (to school) [done]
```

And then this one:

```
root
```

```
..go (to school) [done]
....take_off [done]
....fly_atom (to school) [done]
```

If the policy were to announce the state change of each node, the system would make the following announcement:

(21) S: I have flown to the school. [corresponds to *fly_atom* node]
S: I have flown to the school. [corresponds to *go* node]

Indeed, it would end up making the same announcement twice in a row! In order to avoid this, we might decide to only make an announcement when a *leaf* of the tree changes in state, as *leaves* are the activities which are actually executed. However, this may lead to system announcements which do not contain as much information as they ought to. Consider the situation in which the system is transporting the medical supplies from the hospital to the school. Just before completing this activity, the activity tree looks like this:

```
root
..transport (medical supplies) (from hospital) (to school) [current]
....pick_up (medical supplies) (from hospital) [done]
......go (hospital) [done]
........take_off [done]
........fly_atom [done]
......pick_up_object (medical supplies) [done]
....deliver (medical supplies) (to school) [current]
......go (school) [done]
........take_off [skipped]
........fly_atom (school) [done]
......drop_object (medical supplies) [current]
```

When `drop_object` is completed, it will become *done* and then `deliver` will become *done*, and finally `transport` in turn will become *done*. Here, however, it is desirable to announce not only that the medical supplies have been dropped, but that this indeed concludes the deliver and transport activities – without this information, the operator may become confused as to the state of the robot. The desired dialogue, then, is something like the following:

(22) S: I have dropped the medical kit.
S: I have delivered it to the school.
S: I have transported it from the hospital to the school.

In order to allow for this type of dialogue, the policy which has been implemented is that the completion of an activity is announced under the following conditions:

1. The activity is a leaf node
2. The activity is not a leaf, but it has a different *Natural Language Mapping* from its last child.

8.4.2 Filtering Against the State of the World

The CSLI Dialogue Manager makes use of a relatively common technique in dialogue managers that deal with complex systems in that it employs a *generation manager* which stores potential system utterances in a queue and then, when an appropriate point in the conversation arises for the system to make an utterance (or an utterance is of a critical enough nature that the system should barge in and utter it, no matter what), the generation manager chooses an appropriate utterance from the queue and utters it.[8] In the CSLI Dialogue Manager, this queue is referred to as the *System Agenda*.

Whenever the dialogue manager makes the decision that it is appropriate to announce the state of an activity, using the algorithm discussed in (8.4.1) above, the utterance describing the particular activity and its state is added to the *System Agenda*. Eventually, the generation component will have a chance to examine the *System Agenda* and decide how to realize each utterance linguistically, and whether or not the utterance should be actually be uttered by the system at all. In the situations that the Activity Modeling System described in this paper has been designed for, it's possible that by the time it's appropriate for the generation component to realize an utterance on the System Agenda, this utterance may no longer be relevant – or worse, it may actually represent a claim that is no longer true. Such time delays have tended to occur on a regular basis with the dialogue systems we have developed, mainly because in the seconds it takes for either the operator or the system to make one or a few utterances, the activities being monitored may have changed significantly in nature.

[8]This is a very brief overview of how the generation component in the CSLI Dialogue Manager works; geneeration is, in fact, relatively complex. For example, it will introduce anaphora and ellipsis into the utterances in its queue, in order to fit them better into the current conversation. Such functionality has no bearing on the discussion here, as the algorithm discussed here will benefit any dialogue management system which makes use of a queue to store potential utterances.

A quite common example is that while sometimes it may take a device a few seconds (or even minutes) in order to *plan* an activity, sometimes this process is nearly instantaneous. However, during the instant when the activity switches to a state of `sent` (and is sent to the planner), the dialogue system doesn't automatically know if this state will take a few minutes, seconds, or milliseconds. As such, it immediately places a logical form for an utterance of the form *Now planning to X* on the System Agenda, where *X* is a description of the activity in question. It may be, however, that just a few milliseconds later, before the generation component has even started to process the System Agenda,[9] that the state of the activity in question is changed to *current* by the planner, which has made its plan and begun executing the activity. In this case, when the generation component processes the system agenda, it will be inaccurate for it to announce *Now planning to X*, because it's actually the case that the system is *currently doing X*.

The top utterance on the System Agenda, then, is not really reflective of the state of the device. Indeed, if the system were to utter it, *mode confusion* would surely arise. As such, the generation component of the CSLI Dialogue Manager employs a *filtering* mechanism, in which all utterances regarding a particular activity are checked against the current state of that activity on the Activity Tree before they are uttered. If the utterance describes the activity as being in a state which is no longer correct, then the utterance is filtered out – it is discarded, never actually uttered by the system. According to this algorithm, the above problem is solved by discarding the logical form corresponding to *Now planning to X* and then later actually realizing an utterance like *Now Xing*.

The real power of this approach can be seen by examining a more complex example from the WITAS domain, in which the device (the helicopter) changes its state quite quickly and rather significantly. Consider the case in which the operator gives the command: *fly to the school and look for a red car*. Now imagine that right after the helicopter takes off and just as it begins flying to the school, it sees a red car. In the WITAS domain, the activity of *find* which corresponds to the operator's command of *look for*, specifies that once an object

[9] In the CSLI Dialogue Manager, the generation component tends to wait for the system to "settle down" before it processes the System Agenda – that is, it tends to wait for a series of swift updates about the state of the device to complete before it processes the System Agenda

matching the description given by the operator is spotted, it should be tracked (kept in view). However, in order to do this, the helicopter must suspend flying to the school, since it can't both track the car and fly to the school at the same time. To sum up, the Activity Tree (abbreviated below), goes through the following configurations:

(23) *Taking off and looking for a red car*
Activity Tree:

```
root
..go (to tower) [current]
....take_off [current]
..locate (red car) [current]
```

System Agenda: *empty*

(24) *Finished taking off, started to fly to tower*
Activity Tree:

```
root
..go (to tower) [current]
....take_off [done]
....fly_atom (to tower) [current]
..find (red car) [current]
....locate (red car) [current]
```

System Agenda:

1. *I have taken off*
2. *Now flying to the tower*

(25) *Spotted a red car, started tracking it and suspended flying* (occurs **before** utterances on System Agenda in (24) are processed)

Activity Tree:

```
root
..go (to tower) [suspended]
....take_off [done]
....fly_atom (to tower) [suspended]
..find (red car) [current]
....locate (red car) [done]
....track (red car) [current]
```

System Agenda:

1. *I have taken off*
2. *Now flying to the tower* [FALSE!]
3. *I have found a red car*
4. *I have suspended flying to the tower*
5. *Now tracking the red car*

Without a representation of the current state of the device, the generation component would simply "read off" the system agenda, making announcements about the state of the world which are no longer true (in this case, that the helicopter is currently flying to the tower). The filtering algorithm given above, however, requires that the system skip the announcement that it is currently flying to the tower – since this statement no longer accurately represents the state of the activity. After hearing this utterance, the operator might be quite confused since on the GUI he or she could observe a red car and helicopter motion that appears to indicate that the helicopter is following the car, rather than flying to the school.[10] As such, it becomes apparent that the Activity Tree provides a means for the generation manager to discard or modify reports that have been produced by the system. The dialogue manager can retain the modularity provided by the fact that the generation module is separate from the report-

[10]Perhaps, rather that simply throwing away the utterance *Now flying to the tower*, an even cleverer generation algorithm might change it to something like *I was flying to the tower, but I've now suspended flying there.* No changes would be need to the Activity Tree or the Report Generation Mechanism to facilitate this, only to the Generation Component

generating module, while at the same time having the ability to make sure that the reports the system utters are actually *true*.

8.4.3 Answering *Why?*

Due to the complexity of some activities (in their many sub-activities, sub-sub-activities, and so on) and the length of time it takes to do a particular activity, it may not always be immediately apparent to the operator why the system is doing a particular action. The operator may simply have forgotten that he or she gave a particular command, or perhaps may not realize that the system is doing a particular activity because the activity is actually a subactivity of another activity. In the WITAS system, for example, there is defined a relatively complex activity called *fight_fire* in which the helicopter repeatedly picks up water at one location, transports the water to a second location where a building is on fire, and drops loads of water there until the fire has been extinguished. Because this activity is relatively complex and has a long duration, it's possible that the operator might want to question the helicopter as to why it is, say, picking up the water from the lake.

At at least a simple level, the Activity Tree offers a straightforward means of answering such *why* questions. In order to answer why the device is doing a particular activity, the dialogue manager can look at the activity's ancestor nodes on the Activity Tree and simply report an appropriate ancestor. For example, in (26) there appears a snapshot of the Activity Tree as it might appear during one stage of fighting the fire at the school – specifically, the point at which the helicopter has picked up the water and is carrying it to the school.

(26)
```
root
..fight_fire (at school) [current]
....transport (water from lake to school) [current]
......pickup_at_location (water from lake) [done]
........go (to lake) [done]
.........take_off [done]
.........fly_atom (to lake) [done]
........pickup_object (water) [done]
......deliver (water to school) [current]
........go (to school) [current]
.........take_off [skipped]
.........fly_atom (to school) [current]
```

Given this Activity Tree, the CSLI Dialogue Manager supports such queries as the following:

(27) a. Why?

b. Why did you pick up the water at the lake / go to the lake / take off / pick up the water?

c. Why are you delivering the water to the school / going to the school?

In order to answer each of these questions, the dialogue manager must first determine which activity specifically the user is asking a *why* question about. Once this has been determined, it must choose the appropriate ancestor of this activity to report as an answer to the question. Most of the time, this is simply the parent of the activity in question. There is one case, however, in which the parent is not an appropriate response – namely, the case in which a report in natural language describing the parent activity is identical to one describing the child. For instance, if the activity being asked about is *fly_atom*, then it is inappropriate to report the *fly_atom*'s parent activity, *go*, because both *fly_atom* and *go* are realized linguistically in the same way (in the above Activity Tree, both are realized as something such as *I have flown to the lake*). As such, if the system were to decide that *fly_atom* were the relevant activity and then simply report its parent, infelicitous dialogues like the following could occur:

(28) O: Why did you fly to the lake?
S: #Because I was flying to the lake.

whereas the 'appropriate' exchange should be the following:

(29) O: Why did you fly to the lake?
S: Because I was picking up the water at the lake.

In order to answer *why* questions like the ones in (27) the dialogue manager uses the algorithm given below. Note that the input to the algorithm is a *logical form* representing a *why_query*. It is assumed that the format of the logical form is the following:
`why_query(ActivityMarker, ActivityDescription)`
where `ActivityMarker` is can have one of the following values:

- `anap`: for the purely anaphoric utterance of *why?*
- `currActivity`: for utterances referring to the current activity, either *Why are you Xing?* or *Why are you doing that?*

- `complActivity`: for utterances referring to a completed activity, either *Why did you Xing* or Why did you do that?

and `ActivityDescription` has either the value of `anap` for utterances that don't refer to a specific activity (e.g. *Why?* and *Why are you doing that?*), or the logical form for commanding an activity that can be decomposed by the machinery developed in section 8.1.

```
Algorithm:  ANSWER WHY QUERY
Given:  The logical form w of a why_query
a = find_relevant_activity(w)
r_a = generate_logical_form(a)
p = parent(a)

while(p != null) {
    r_p = generate_logical_form(p)
    if(r_p != r_a) return why_answer(r_p)
    p = parent(p)
}
```

And the algorithm for `find_relevant_activity` is as follows:

```
Algorithm:  FIND RELEVANT ACTIVITY
Given:  The logical form w of a why_query with ActivityMarker m and
ActivityDescription d
Given:  The list of salient activities, S
if m = anap AND d = anap return first(S)
foreach s in S {
   a = parse_command_to_activity(d)
   if matches(s, (a, m)) return s
}
```

where the `matches` predicate takes an activity as one parameter and a description of an activity with its state as the second parameter and returns true if and only if the description and state of the second activity are the same as the first.

8.4.4 Answering *What are your constraints?*

Since the system allows the operator to specify rather complex sets of constraints on the device, it becomes immediately important for the the operator to be able to find out from the system what exactly it believes its current set of constraints is. Given that the dialogue interface already must be capable of converting constraints into nat-

ural language, answering the question *What are your constraints?* is fairly simple. Currently, the dialogue manager simply reads off all of the *necessary* and *banned* constraints on the *root* node of the Activity Tree to produce dialogues like the following:

(30) O: Always fly high.
S: Okay.
...
O: Never fly at low speed.
S. Okay.
...
O: What are your constraints?
S: I am supposed to always fly high.
S: I am never supposed to fly at low speed.

Granted the output is fairly simplistic. For more natural output, the Generation Component should probably aggregate these utterances into a single utterance. This is relatively straightforward and planned as future work.

9 Limitations and Future Work

While this paper identifies major steps which can be taken toward designing generic dialogue systems which are capable of facilitating task-oriented dialogues, it certainly doesn't offer a complete, flawless solution to the problem. There are a number of problems which simply haven't been addressed and some which haven't been addressed completely. In this section, I'll discuss some of these issues and try to mention ways in which the framework presented here might provide useful insights or a first step toward solving them.

9.1 Grammar Development and Speech Recognition

Throughout most of this paper, the process of first converting an acoustic signal representing spoken language input into text and then parsing this text into some sort of *logical form* the dialogue manager can use has been largely taken for granted. Many dialogue systems, including the ones developed at CSLI [LGP02] and NASA [RHJ00],

use a domain-specific *grammar* to parse text input (and often bi-directionally to produce text output). Often this grammar is compiled into a *language model* which an automatic speech recognizer (ASR) uses as a constraint on the utterances it expects to hear. Other systems might use corpus data or other statistical techniques to constraint or train their speech recognizer. At the moment, the grammars and language models used by most dialogue systems are highly domain-specific. For example, the one used by the CSLI dialogue manager for the WITAS system is specific to the types of utterances that are involved in controlling an autonomous helicopter. As such, it would be totally incapable of parsing utterances related to tasks like driving a car or controlling a radio.

The question, then, is whether or not a domain-independent grammar can be written which is suitable across a large number of conversational domains. It might be possible to directly plug such a grammar into many different dialogue systems, or it might be necessary to *specialize* it in some, relatively straightforward way, across many applications. While the work presented in this paper clearly doesn't answer the question of how such a grammar could be implemented, I believe that it sheds some light on the process.

The framework provided here identifies many of the common types of conversations which are likely to occur as part of task-oriented dialogues. As such, it provides some guidance in the *range* of utterances that a general-purpose grammar would have to provide, assuming the grammar was to be geared only toward task-oriented, practical dialogue systems. By identifying common classes of dialogues, we provide a metric by which a general-purpose, domain-independent grammar could be measured; we could, for instance, count the number of domains in which a specific grammar supports the range of dialogue facilitated by the framework presented here.

Moreover, the framework given here provides an explicit, domain-independent (across task-oriented domains) semantic mechanism for representing many sorts of utterances. For instance, it provides a generic way to semantically specify activities – as sets of required and optional definable slots – as well as constraints over these activities – as logical expressions over the values of these constraints. If we were to design a domain-independent grammar which also used representations compatible with those given here, then we would have an extremely straightforward means of specializing this grammar to specific domains: we could, for example, define mappings between

components of an utterance and various slots relevant to the domain in question. Indeed, it seems relatively straightforward to imagine adding extra fields to the recipe script for each activity which act essentially as sub-categorization mappings between arguments to a verb and relevant slots.

9.2 More Complex Recipes

Throughout this paper it has often been assumed that any recipe which we would want to describe could be effectively and easily described through the recipe script.[11] This, of course, is not the case as a recipe could, in theory, be an arbitrarily complex set of instructions. It does seem, however, that there are several concepts not included in the capabilities of the recipe scripting language which are useful across a large range of task-oriented dialogues.

One major issue is handling goal-oriented decomposition. In this simplest case, this manifests itself as a choice between two different ways of accomplishing the same goal. Imagine, for instance, that you want me to be at your house for a party at 11:00 and that I have several different possible ways that I could get there; for instance, I could fly, drive, walk, bicycle, or skateboard there. Now imagine, further, that you don't care *how* I get there, just that I am indeed at your house at some point around 11:00. In this case, I might choose any of the above options in order to get to your house. Of course the situation can rapidly get more complicated. It might be that you live too far away for me to skateboard or walk, and that I don't have access to a small plane or a helicopter, so I'll have to drive or bicycle to the party. Or perhaps I get on my bicycle, but just as I'm leaving my house, the peddle breaks off and suddenly I have to drive to your house instead.

What I've identified here essentially is goal-oriented, rather than task-oriented, decomposition. While the recipe scripting language outlined in this paper allows for activities to be decomposed into specific sequences (or simultaneous sets) of other activities, this perhaps doesn't quite mirror the way in which humans actually decompose activities. Humans are often quite flexible in that if one way of accomplishing a goal fails, they'll try a different way (the broken bicycle). Or if for particular reasons they can't even try one means of achieving

[11] Special thanks to the members of the Stanford Natural Language Processing reading group for an interesting discussion on the issues mentioned in this section. Any errors are, of course, mine.

a goal, they try another means instead of giving up (your house is too far away, and I don't have my own helicopter). Moreover, they might defer making decisions about how to accomplish certain goals until a later date when they have more information.

It is perhaps more appropriate to say, then, that humans naturally decompose recipes in terms of *goals* for which they already know (or can learn about, discover, or invent) recipes for means of achieving these goals. Such decomposition is not supported by the recipe scripting language at the moment, not because it is more difficult to *write* scripts in this way, but mainly because the machinery involved in *executing* scripts so decomposed is more demanding. Indeed, the Activity Tree and the Constraint Management System are agnostic as to whether recipes are decomposed in terms of goals or other recipes – the tree simply represents planned, current, and past actions no matter by what mechanism they were generated, and the constraint system applies recipes being instantiated for any reason. Moreover, syntacticly it would be relatively straightforward to specify *goals* instead of specific recipes which should be *intended* within a recipe body. However, it is exactly the process of making the decisions regarding which recipe to use to achieve a specific goal and why, that I sought to avoid in the framework provided here. Such decisions may require rather complex planning and real-time execution systems on the part of the device being controlled – a requirement I didn't wish to impose on the systems which are being dialogue-enabled for the moment. Instead, I chose to focus on a wider-range of systems which might or might not have such a component. This is not to say that such systems should be forever ignored; indeed, I believe that much interesting useful work can be put into means of generically dialogue-enabling the features of such systems. For example, dialogues about which recipe to use and why may surely share commonalities across many devices that could be captured and added to the framework described here.

Indeed, I believe that the framework presented in this paper could be expanded in a straightforward manner to deal with goal-decomposition as opposed to activity-level decomposition. Certainly, the Activity Tree and constraint management systems as they exist would work fine with such a system. What would remain would be to implement algorithms for facilitating dialogues regarding which solutions are under consideration and why and for actually picking a solution to execute. Moreover, the constraint system would have to be expanded to deal with constraints which describe which recipe to choose

to fulfill a particular goal. The work presented in this paper serves as a good basis for such expansion, and it provides a framework for future expansion because many useful notions are already represented and realized computationally. Already in place is a semantics for goals in which they are expressed in terms of slots, which are used to conceptualize activities; such a semantics could certainly be utilized in a goal-decomposition system. In addition, the Activity Tree could be "multiplied" – that is, rather than having a single Activity Tree which represents that actual state of the device, Activity Trees which represent "possible worlds" could be created in order to facilitate discussions about different possibilities for how to pursue a particular goal. Once a "possible world" is decided upon, the activity tree representing that world could be attached as a subtree of the one which represents the "actual" world.

In the meantime, devices with such capabilities can still be interfaced to the existing system. At the level where such decisions should be made, recipes should simply be declared as *atomic* and be sent directly to the device for planning. At this point, the device has the freedom to choose whatever set of actions it wishes to take. If such actions will be relevant to the dialogue, then the device is free to represent these actions as activities on the Activity Tree, which appear below what would otherwise be an atomic leaf node.

9.3 Natural Language Descriptions of Recipes

Given that the system has recipes which describe how to accomplish certain goals, it would be natural to allow it to discuss (and even possibly modify) these recipes using natural language with the human operator. Indeed, this goal was one motivation for keeping the recipes relatively simple in nature. There are potentially two different levels at which a recipe might be discussed. In the first case, the recipe would be discussed in purely abstract terms, while in the second, it would be at least partially instantiated.

The difference is most evident in that it would manifest itself in the answers to the following two different questions we might like to ask the system:

1. How do/would you patrol? / How does one patrol? / What's involved in patrolling?
2. How do/would you patrol between the tower and the school?

In the first case, the response should really involve the elements of the recipe for patrolling. A suitable response might be something close to *In order to patrol, one must continually fly to one location, then a second location.* In the second case, the situation is much more concrete and the answer could be suitably more concrete. It might take the form of *In order to patrol between the tower and the school, I would continually fly to the tower then to the school.*

In the first case, several complexities emerge which must be dealt with. The first is actually determining precisely which recipe the human operator wants to talk about. As a first go, we might assume that we would simply do a reverse lookup according to the NL mapping of recipes in the library; that is, we would simply search for the recipe whose NL mapping matches the verb being asked about. Such a reverse lookup runs into the immediate problem that there may actually be *several* recipes which map to the same verb (since this phenomenon is, in fact, the very reason that the NL mapping system was created – please see section 5.2.1). In this case, a question like *How would you patrol?* may involve several possible answers (in the WITAS system), since the verb of *patrol* actually maps on to multiple concepts.[12] If such multiple matches were found, then the system would have to either decide all of them, use some sort of probabilistic means or weighting schema to decide which one to say, or initiate a clarification subdialogue to try to determine which one, specifically, the human operator is interested in.

Once the relevant recipe has been isolated, its *recipe body* must then be described (and perhaps, its goals, preconditions, constraints, and so on if so desired). In order to do this, an algorithm would have to be designed which could examine the recipe body script and produce reasonable natural language to describe it. Some of this would involve natural language constructs to describe loops – as the use of "continuously" above illustrates. The main difficulty which would emerge, I believe, is when dealing with the question of how to best describe the uninstantiated recipes using natural language. Perhaps the most

[12]These concepts are the following:

- patrolling among multiple locations
- patrolling at a particular location
- patrolling among various locations while looking for a specific object
- patrolling at a particular location while looking for a specific object

straightforward solution would be to associate some sort of phrase with each slot type (or perhaps at a finer grain, with each slot definition or even each slot declaration within each recipe) which could be used to describe it abstractly. For instance, imagine that associated with the type `Location` in the WITAS system was some noun phrase like *a location*. Then, in order to generate the description of an invocation of a recipe like *fly*, we could simply fill in *a location* where we would usually fill in the NP corresponding to the object in the `toLocation` slot, yielding something like *Fly to a location*. Of course, such a system immediately shows its limitations when confronted with the question about *patrolling* above; such simple replacement rules would yield an answer similar to *I would continually fly to a location then fly to a location*. Clearly, at some point, a relatively sophisticated generation algorithm would have to be used to avoid such obviously bad generation.

We see then, that by limiting the constructs in the recipe scripting language, we could envision a system that could describe these recipes in abstract terms using natural language. While building such a generation algorithm would clearly be non-trivial, I have sketched here the major considerations that would have to go into it. The next issue, then, is how to answer questions like the second one: *How do/would you patrol between the tower and the school?*. In some sense, this is a much more difficult problem since it involves analyzing the current context. For instance, if the helicopter were currently at the tower, then the answer given above wouldn't seem quite correct – indeed, we would want to say that the helicopter would first fly to the *school* and then to the *tower*, since this is the order in which it would actually do things, given the current state of the world. In this sense, this is a much more difficult problem than then discussing recipes in an abstract sense. Indeed, in order to give a reasonable answer, we are likely to actually want to try to simulate the device actually executing the activity, given the current state of the world as the start state of the simulator. This is the case not only because certain actions might "obviously" be skipped, but also because we want to simulate the effects of the current constraint set on how certain activities would be performed.

10 Conclusions

The work presented in this paper provides evidence for the domain-independence hypothesis described in [ABD+01], repeated here:

> "Within the genre of practical dialogue, the bulk of the complexity in the language interpretation and dialogue management is independent of the task being performed."

Specifically, it describes the implementation of relatively generic dialogue-management *algorithms* which operate over declaratively specified information about a particular intelligent agent/device to yield a conversational system which can be used by a human operator to command and control the agent/device, as well as participate in *joint-activities* with it. Specifically, this is done by defining an interface which lies between the dialogue manager and the agent/device which provides a domain-independent entity for the dialogue manager to work with which is capable of modeling how joint-activities work in general. This interface is then specialized to each agent/device by specifying a recipe library, which defines the specific capabilities of the agent/device.

By writing dialogue-management algorithms which operate in terms of structures on the Activity Tree and constraints in the Constraint Management System, the dialogue manager can be imbued with dialogue strategies which work *in general* across a wide range of agents/devices. That is, by isolating *general* aspects of task-oriented dialogue, it is possible to create a dialogue system that supports many of the classes of task-oriented dialogues. Many issues that arise in task-oriented dialogues were discussed in this paper, and algorithms for solving these issues in the general case were presented. Specifically, the appropriate way to model the following two issues was discussed:

- How to structure, decompose, and conceptualize joint activities (solution: Activity Tree, Recipe Scripts)
- How to model constraints which people are apt to impose using natural language (solution: Constraint Management System)

I was the able to show that with these models in hand, relatively generic algorithms could be introduced to facilitate common task-oriented dialogues. In specific, I considered and proposed solutions for the following issues:

- Using the commonsense knowledge of how activities are decomposed to interpret utterances in context and produce meaningful

(or decide to filter out no-longer-relevant) utterances.

- Dialogues for dealing with conflicts over resource usage
- Algorithms for engaging in the dialogue games which arise when constraints come into conflict with one another, or with defaults.
- Strategies for converting constraints back and forth between natural language and first-order-logic
- Using the structure of the Activity Tree to answer questions like *Why?* and *What are you doing?* in order to avoid mode confusion by clearly communicating the state of the device

The work presented here identifies many of the common genres of conversations which are likely to occur in the pursuit of concrete tasks. It proposes strategies for dealing with such dialogues across a wide range of tasks and task participants. I also identify the current limitations of the system and determine possible ways by which these limitations be addressed in the future by building on the current framework.

A Adapting the Dialogue Manager to a New Domain

In this section, I will briefly describe the technical details involved in supporting a new task-oriented domain for dialogue. In particular, I will assume that the goal is to modify the CSLI dialogue mananger to work with a new device or agent. Ideally, all of the work would need only to be *declarative*, in the sense that the dialogue manager's Java program code shouldn't have to be modified. As I'll describe here, in practice some of the code must indeed be modified. The changes made are fairly routine and straightforward, however, and it is my belief that future work could render them declarative in nature.

The steps involved are as follows:

- A new recipe script for the device needs to be created and compiled.
- The device will need to be interfaced to the recipe executor module.
- Callback methods describing parallel relationships among slots will need to be defined in the dialogue manager.

- The resolution procedures for activities in the dialogue manager may need to be modified slightly to perform domain-specific inferences.
- The current grammar will need to be adapted or rewritten to deal with the device.
- The procedures for converting between the logical forms produced by the grammar and the slots defined in the recipes may need to be modified.
- Databases which supply knowledge about objects in the world will need to be created.

Of immediate note is that none of these steps involve modifying the Constraint Management System or the way in which the Activity Tree functions. Many of the above steps are fairly trivial, while we will see that a few require a significant amount of work.

A.1 Creating and Compiling a Recipe Script

The syntax and layout for the recipe scripting language has already been discussed in great detail in sections 5 and 6. In this section, I will describe the technical details of how to actually 'compile' the script and incorporate it into the dialogue manager as a whole.

Assuming we have an recipe script named `myDevice.ts`, the first step is to 'compile' it into the files needed by the system at runtime. This is done using the `CSLI_RecipeCompiler`, using the following command:

```
: java csli.recipe.CSLI_RecipeCompiler myDevice.ts
```

Note that Java version 1.3 or above should be used. This will generate the following files and place them in a subdirectory of the current directory named `output`:

1. `CSLI_ActivityProperties.java`: defines the activity properties
2. `myDevice.rep` (this file name will actually depend on what it is specified to be called in the recipe script, see section 5.1.1).
3. `CSLI_TaskMatcher.java`: A simple class with a hash table to do NL mapping of command names
4. `domains.ecl`: Defines the domain of each slot

All of these files should then be copied into the following directory of the dialogue manager code:

```
CSLI_HOME/csli/agents/dialogueManager/activityModel/
```

This directory should be recompiled with the following command:

```
cd CSLI_HOME/csli/agents/v2/dialogueManager/activityModel
javac *.java
```

In addition, `myDevice.rep` should be copied to `csli/agents/v2`.

At this point, the dialogue manager has been "adapted" to deal with the recipes specific to this device.

A.2 Interfacing the Device to the Recipe Executor

The recipe executor reads in the recipes, instantiates them into activities, and executes them as needed. When it encounters an *atomic* activity, however, it needs to be able to send this activity to the device to actually be executed. Moreover, as the device executes the activity, it needs to be notified of the changes in the state of the activity (for example, is it `current`, `planned`, `suspended`, and so on). This is done through the `deviceInterface` module, located in `csli/agents/v2/deviceInterface`. Here two relevant Java `interface` specifications are defined: `CSLI_Device` and `CSLI_DeviceListener`.

The first, `CSLI_Device` defines a set of methods that the device must be able to respond to. In order to interface a device to the dialogue manager, a Java class which is capable of responding to calls to these methods by sending information to the actual device must be defined. This may simply be a *stub*, which relays the calls to the "real" interface to the device through CORBA, OAA, RMI, or some other architecture (this is how the CSLI system interfaces to the robotic helicopter). Alternatively, if the control regime for the device is built in Java, then the code controlling the device may simply be modified to implement this interface (this is how the CSLI system interfaces to the *simulator* of the robotic helicopter). The major part of the interface appears in figure 5

The comments in the code describe each method. The methods are in support of the following capabilities:

Figure 5: The bulk of the CSLI_Device Interface

```
/**
 * defines the interface that the device must adhere to
 * for the SimTaskTree to interface with it
 */
public interface CSLI_Device {
    /**
     * add a listener to be notified of device events
     */
    public void addListener(CSLI_DeviceListener listener);

    /**
     * execute an atomic activity
     * @param id the id of the activity
     * @param properties the properties of the activity
     */
    public void executeAtomic(String id,
                              CSLI_ActivityProperties properties);

    /**
     * test the value of a predicate
     */
    public boolean testPredicate(String predicate,
                                 ArrayList arguments);

    /**
     * should be equivalent to cancel(id, true)
     */
    public void cancel(String id);

    /**
     * cancel the activity with the given id
     * @param id the id of the activity to cancel
     * @boolean shouldSetCancelled should be true
     * iff the device should now notify the
     * listeners that the activity has been cancelled
     */
    public void cancel(String id, boolean shouldSetCancelled);
```



```
    public void fillMonitorSlots(CSLI_ActivityProperties ap);
}
```

executeAtomic Takes in the name and parameters of an atomic activity and executes it. A unique identifier is also passed in so that the dialogue manager has a means of communicating with the device about this specific activity.

testPredicate In the recipe script, it is legal to specify predicates as goals, preconditions, and as the conditions of loops in the recipe body; this method is used by the dialogue manager to determine at runtime if these predicates should evaluate to true or false. The device must be able to determine if a given predicate, with a given set of arguments (represented as `String` objects), is true or false when this method is called.

cancel Cancel a specific activity (and optionally notify the listeners).

fillMonitorSlots The monitor slots (see section 5.2.3) must be filled in at runtime when this method is called. In this way, the dialogue manager can ask the device to reflect about its current state, on demand.

addListener The device needs to be able to support the typical Java notion of having a *Listener*. Here, each `CSLI_DeviceListener` object which it is passed (via the `addListener` method), must be notified whenever the state of the device change. This will be discussed immediately below.

While the device must be able to respond to the above methods, it must also have a means of notifying the dialogue manager when the state of an atomic activity has changed (for example, from `planned` to `current`). In order to do this, whenever the state of an activity changes, the device must notify the `CSLI_DeviceListener` objects which have registered with it via the `addListener` method. Figure 6 shows the bulk of the `CSLI_DeviceListener` interface. As would be expected, the interface is concerned mainly with communicating the *state* of activities to the dialogue manager.

A.3 Callback Methods for Effective Slot Lengths

As was mentioned sections 7.4 and 7.4.1, so-called parallel slots sometimes may be useful. For example *toLocation*, *toSpeed*, *toAltitude* in the WITAS system are defined to be parallel slots, since the location that the helicopter flies to must also always be accompanied with a speed and and altitude at which to fly. In pursuit of supporting these

Figure 6: The bulk of the CSLI_DeviceListener Interface

```
/**
 * the device should call these methods on its listeners
 */
public interface CSLI_DeviceListener {
    /**
     * called when a task is completed
     */
    public void taskCompleted(String taskID);
    /**
     * called when a task is planned
     */
    public void taskPlanned(String taskID);
    /**
     * called when a task is cancelled
     */
    public void taskCancelled(String taskID);
    /**
     * called when a task becomes a current task the uav is working
     * on
     */
    public void taskCurrent(String taskID);
    /**
     * called when a task fails
     */
    public void taskFailed(String taskID);
    /**
     * called when a request to stop tasks on the list
     * is made
     * @param taskIDs the ids of the tasks to stop
     */
    public void stopTasks(java.util.ArrayList taskIDs);
    /**
     * called when a request to stop all tasks
     * has been made
     */
    public void stopAllTasks();
    /**
      * @param value if it is true, the the recipe executor
      * should be planning and executing recipes
      */
    public void setShouldPlan(boolean value);
}
```

parallel notions, the following callback methods must be defined by the dialogue manager for a specific device:

```
int getSlotMinLengthForDefault(String slotName,
                               CSLI_ActivityProperties ap)

int getSlotMaxLengthForConstraint(String slotName,
                                  CSLI_ActivityProperties ap)
```

Note that each method takes a `slotName` and a `CSLI_ActivityProperties` object. The first parameter specifies the name of the slot in question, and the second is essentially a list of all the slots paired with their value lists. So, for example, if `slotName` were the String `toSpeed` and in `ap` the slot `toLocation` had 2 values filled in, then both methods would return the value 2.

The dialogue manager provides default behavior for each of these methods in the `CSLI_ActivityBase` class in the following package:

`csli.agents.v2.dialogueManager.activityModel`.

But it should be overridden in the case of parallel slots in the subclass, `CSLI_Task` which is used by the dialogue manager.

A.4 Resolution Procedures for Activities

The dialogue manager needs to know when a particular partially specified activity should be set to the state `resolved` (and hence then subjected to the algorithms which attempt to use constraints and defaults to fully instantiate it). In general, this means that all of the *required definable slots* in the activity must have a value (indeed, each required definable slot must at least contain a list of values equal in length to its declared minimum length). Usually these values come directly from user utterances, however sometimes they can be filled in through *inferences*. For example, in the WITAS system, if the helicopter is told to *deliver medical supplies*, it needs to know where the medical supplies are in order to pick them up. If there is only one set of medical supplies, and it is known to be at *Springfield Hospital*, then the system should infer that it should fly to the *Springfield Hospital* to pick them up, without having to ask for this required definable slot to be filled in by the user. This sort of inference, in general, is domain specific. As such, in the `CSLI_TaskHelper`, a callback method is

defined called `tryResolving` which takes a partially specified activity and tries to make inferences to fill in its unfilled slot values. If the new device requires any such inferences, then this method should be redefined in `CSLI_TaskHelper` and the dialogue manager should be recompiled.

A.5 Modifying the Grammar and the Conversion Routines

This topic has already been covered in section 9.1. Suffice it to say that the grammar must be adapted to a new domain. It would be desirable if the there were a domain-independent grammar that could be specialized for each new device, but this has yet to be developed.

A.6 Creating New Databases

Currently, the CSLI Dialogue manager currently requires two databases to represent real objects in the world. They are written using Knowledge Interchange Format, and are searched using Stanford's Java Theorem Prover [jtp]. Logical axioms are used to define hierarchical "isa" relations (for example, the base is a building which is a geographical object). The first database defines the *static* objects in the world, while the second is used to define the *dynamic* objects. In the WITAS system, this is a distinction between things that appear on maps – like roads and buildings – and things that the helicopter sees and reports in real time – like cars and trucks. Noun phrases are then *resolved* and *bound* to specific objects that appear in these databases when the dialogue manager seeks to determine what a given noun phrase refers to.

A new system may use the existing structure of this database, but needs to define a new set of static objects which are salient to the device. For instance, a robot for the home might need to know about the various rooms in the house.

B An Example Recipe Script

The following is the recipe script used in the WITAS system for interfacing a robotic helicopter to the dialogue system.

```
device package csli.agents.v2.simulator;
```

```
dialogue package csli.agents.v2.dialogueManager.activityModel;
//dialogue package csli.recipe; //temp for testing

repfile "witas.rep";

//sets valid atoms for slot values
Types
{
        Location :: ["t1", "s1", "b4", "b6", "b7", "b8", "b9", "b10",
                     "ts1", "b2", "b3", "b5", "b11", "h3", "h4",
                     "h2", "h5", "h7", "r1", "r2", "r3", "r4",
                     "r5", "r6", "r7", "r8", "r9", "r10", "r11",
                     "r12", "r13", "r14", "r15", "r16", "r17",
                     "r18", "r19", "r20", "r21", "w1", "w2", "w3",
                     "f1", "m1", "h6", "p1", "rr1",
                     "waypoint1", "waypoint2", "waypoint3",
                     "waypoint4", "waypoint5", "waypoint6",
                     "waypoint7", "waypoint8", "waypoint9",
                     "waypoint10"];

        Speed           :: ["high", "medium", "low", "zero"];
        Altitude        :: ["high", "medium", "low", "zero"];
        Object          :: []; //[] means it's not involved in constraints
        MoveableObject  :: [];

}

DefinableSlots
{
        Location toLocation:1-3;
        Location fromLocation:1-3 = "null";
        MoveableObject carryObject:1;

        Speed toSpeed:1-3 = "medium";
        Altitude toAltitude:1-3 = "medium";
        Speed fromSpeed:1-3 = "medium";
        Altitude fromAltitude:1-3 = "medium";

        Object searchItem:1;
        Object followItem:1;
```

```
}

MonitorSlots {
        Location curLocation:1;
        Speed curSpeed:1;
        Altitude curAltitude:1;
        Object grippedObject:1;

        //helicopter can only see a single item at a time
        Object noticedItem:1;

}

Resources {
        uav;
        gripper;
        camera;
}


abstract taskdef<move,"move">
{
        DefinableSlots {
                required toLocation;
                optional fromLocation;
                optional toSpeed;
        }

        MonitorSlots {
                curLocation;
                curSpeed;
        }

        Resources {
                uav;
        }


        Banned {
                toSpeed == "zero";
```

```
        }
}

taskdef<go,"go"> extends move
{
        DefinableSlots {
                optional toAltitude;
        }

        MonitorSlots {
                curAltitude;
        }

        Banned {
                toAltitude == "zero";
        }

        NLSlots {
                default: toLocation;
                current: toLocation, toAltitude, toSpeed;
        }

        Body {
                intend take_off(toAltitude = THIS.toAltitude);
                foreach toLocation t, toAltitude a, toSpeed s {
                   intend fly_atom(toLocation = t,toAltitude=a, toSpeed=s);
                }
        }
}

taskdef<fly_atom,"go"> extends move
{
        DefinableSlots {
                optional toAltitude;
        }

        MonitorSlots {
                curAltitude;
        }
```

```
        PreConditions {
                curAltitude[0] != "zero";
        }

        Goals {
                curLocation[0] == toLocation[0];
        }

        NLSlots {
                default: toLocation;
                current: toLocation, toSpeed, toAltitude;
        }

}

taskdef<take_off,"take_off">
{
        DefinableSlots {
                optional toAltitude;
        }

        MonitorSlots {
                curAltitude;
        }

        Resources {
                uav;
        }


        Goals {
                curAltitude[0] != "zero";
        }

        NLSlots {
                default: ;
        }
}
```

```
taskdef<land,"land">
{
        DefinableSlots {
                required toLocation;
        }

        MonitorSlots {
                curLocation;
        }


        Resources {
                uav;
        }

        Goals {
                curLocation[0] == toLocation[0];
                curAltitude[0] == "zero";
        }

        NLSlots {
                default: toLocation;
        }

        Body {
                intend go(toLocation = THIS.toLocation);
                intend land_atom();
        }
}

taskdef<land_atom,"land">
{
        DefinableSlots { }

        MonitorSlots {
                curAltitude;
        }

        Resources {
                uav;
```

```
        }


        PreConditions {
                curAltitude[0] != "zero";
        }

        Goals {
                curAltitude[0] == "zero";
        }

        NLSlots {
                default: ;
        }
}

taskdef<patrol_between_search,"patrol">
{
        DefinableSlots {
                required toLocation;
                required searchItem;
                optional toAltitude;
                optional toSpeed;
        }

        MonitorSlots {
                curLocation;
                curAltitude;
                curSpeed;
        }

        Resources

                uav;
        }

        PreConditions

        }
```

```
        Goals

                refers_to(searchItem[0], noticedItem[0]);
        }

        NLSlots

                default: searchItem;
                current: searchItem, toLocation, toAltitude, toSpeed;
        }


        Body

                intend patrol_between(toLocation = THIS.toLocation,
                                      toAltitude=THIS.toAltitude,
                                      toSpeed = THIS.toSpeed) noblock;
                intend find(searchItem = THIS.searchItem);
        }
}


taskdef<find,"find"> {
        DefinableSlots

                required searchItem;
        }

        MonitorSlots

                noticedItem;
        }

        PreConditions

        }

        Goals

                noticedItem[0] == searchItem[0];
```

```
        }

        NLSlots

                default: searchItem;
        }


        Body
        {
                do {
                    intend locate(searchItem = THIS.searchItem);
                    intend track(followItem = THIS.noticedItem) t noblock;
                    intend identify(searchItem = THIS.searchItem);

                    stop t;
                } while(not_refers_to(searchItem, noticedItem));

                intend follow(followItem = THIS.noticedItem);
        }
}

taskdef<locate,"find"> {
        DefinableSlots {

                required searchItem;
        }

        MonitorSlots

                noticedItem;
        }

        NLSlots

                default: searchItem;
        }

}
```

```
taskdef<track,"track"> {
        DefinableSlots {

                required followItem;
        }

        MonitorSlots

                noticedItem;
        }

        Resources

                uav;
        }


        PreConditions

        }


        Goals

        }

        NLSlots

                default: followItem;
        }

}


taskdef<follow,"follow"> {
        DefinableSlots {

                required followItem;
        }
```

```
        MonitorSlots

                noticedItem;
        }

        Resources

                uav;
        }


        PreConditions

        }


        Goals

        }

        NLSlots

                default: followItem;
        }

}


//should spawn a dialogue act: "Is noticedItem == item"
taskdef<identify,"identify"> USER {
        DefinableSlots {

                required searchItem;
        }

        MonitorSlots

                noticedItem;
        }
```

```
        PreConditions

                noticedItem[0] != null;
        }

        Goals
        { }

        NLSlots

                default: searchItem;
        }

}

taskdef<patrol_between,"patrol"> {
        DefinableSlots {
                required toLocation;
                optional toAltitude;
                optional toSpeed;
        }

        MonitorSlots {
                curLocation;
                curAltitude;
                curSpeed;
        }

        Resources {
                uav;
        }

        NLSlots

                default: toLocation;
                current: toAltitude, toSpeed;
        }


        Body
```

```
        {
                repeat {
                   foreach toLocation p, toAltitude a, toSpeed s {
                     intend go(toLocation = p,toAltitude=a, toSpeed=s);
                   }
                }
        }
}

taskdef<patrol,"patrol"> {
        DefinableSlots {
                required toLocation;
                optional toAltitude;
                optional toSpeed;
        }

        MonitorSlots {
                curLocation;
                curAltitude;
                curSpeed;
        }

        Resources {
                uav;
        }

        NLSlots {
                default: toLocation;
                current: toLocation, toSpeed, toAltitude;
        }


        Body
        {
                intend go(toLocation = THIS.toLocation,
                          toAltitude=THIS.toAltitude,
                          toSpeed=THIS.toSpeed);
                intend patrol_atom(toLocation = THIS.toLocation,
                                   toAltitude=THIS.toAltitude,
                                   toSpeed=THIS.toSpeed);
```

```
        }

}

taskdef<patrol_atom,"patrol"> {
        DefinableSlots {
                required toLocation;
                optional toAltitude;
                optional toSpeed;
        }

        MonitorSlots {
                curLocation;
                curAltitude;
                curSpeed;
        }

        Resources {
                uav;
        }

        NLSlots {

                default: toLocation;
                current: toLocation, toAltitude, toSpeed;
        }

}

taskdef<patrol_search,"patrol"> {
        DefinableSlots {
                required toLocation;
                required searchItem;
                optional toAltitude;
                optional toSpeed;

        }

        MonitorSlots {
                curLocation;
```

```
                curAltitude;
                curSpeed;
                noticedItem;
        }

        Resources {
                uav;
        }

        NLSlots {
                default: toLocation, searchItem;
                current: toLocation, searchItem, toAltitude, toSpeed;
                conflicts: searchItem;
        }


        Body
        {
                intend patrol(toLocation = THIS.toLocation,
                              toAltitude=THIS.toAltitude,
                              toSpeed = THIS.toSpeed) noblock;
                intend find(searchItem = THIS.searchItem);

        }
}


taskdef<pick_up_object,"pick_up"> {
        DefinableSlots {
                required carryObject;
        }

        MonitorSlots {
                grippedObject;
        }

        Resources {
                gripper;
        }
```

```
        PreConditions {
                gippedObject[0] == null;
        }


        Goals {
                sameid(grippedObject[0], carryObject[0]);
        }

        NLSlots {
                default: carryObject;
        }
}

taskdef<pick_up,"pick_up"> extends go
{
        DefinableSlots {
                required carryObject;
        }

        MonitorSlots {
                grippedObject;
        }

        Resources {
                gripper;
        }


        PreConditions {
                grippedObject[0] == null;
        }


        Goals {
                sameid(grippedObject[0], carryObject[0]);
        }

        NLSlots {
```

```
                default: carryObject, toLocation;
                conflicts: carryObject;
        }

        Body
        {
                intend go(toLocation = THIS.toLocation);
                intend pick_up_object(carryObject = THIS.carryObject);
        }

}

taskdef<drop_object,"drop"> {
        DefinableSlots {
                required carryObject;
        }

        MonitorSlots {
                grippedObject;
        }

        Resources {
                gripper;
        }


        PreConditions {
                sameid(grippedObject[0], carryObject[0]);
        }

        Goals {
                grippedObject[0] == null;
        }


        Banned {
                toAltitude == "medium";
                toAltitude == "high";
        }
```

```
        NLSlots {
                default: carryObject;
        }




}

taskdef<deliver,"deliver"> extends go {
        DefinableSlots {
                required carryObject;
        }

        MonitorSlots {
                grippedObject;
        }

        Resources {
                gripper;
        }


        PreConditions {
                sameid(grippedObject[0], carryObject[0]);
        }


        Goals {
                grippedObject[0] == null;
                at(carryObject[0], toLocation[0]);
        }


        Banned {
                toAltitude == "medium";
                toAltitude == "high";
        }

        NLSlots {
```

```
                default: carryObject, toLocation;
                current: carryObject, toLocation, toSpeed, toAltitude;
        }

        Body
        {
                intend go(toLocation = THIS.toLocation,
                          toAltitude = THIS.toAltitude,
                          toSpeed=THIS.toSpeed);
                intend drop_object(carryObject = THIS.carryObject);
        }

}

taskdef<transport,"transport"> extends pick_up
{
        //toSlots and carrayObject are from pick_up
        DefinableSlots {
                required fromLocation;
                optional fromSpeed;
                optional fromAltitude;
        }

        MonitorSlots { }

        Resources {
                gripper;
        }


        Goals {
                at(carryObject[0], toLocation[0]);
        }

        Banned {
                toAltitude == "medium";
                toAltitude == "high";
        }

        NLSlots {
```

```
                default: carryObject, toLocation;
                current: fromLocation, toLocation, carryObject,
                         toSpeed, toAltitude;
        }


        Body
        {
                foreach fromLocation f, fromSpeed s, fromAltitude a {
                        intend pick_up(toLocation = f,
                                       fromSpeed = s,
                                       fromAltitude = a,
                                       carryObject = THIS.carryObject);
                }

                foreach toLocation t, toSpeed s, toAltitude a {
                        intend deliver(toLocation = t, toSpeed = s,
                                      toAltitude = a,
                                      carryObject = THIS.carryObject);
                }
        }
}

taskdef<fight_fire,"fight_fire"> extends transport
{
        //all from super
        DefinableSlots { }

        /all from super
        MonitorSlots { }

        Resources {
                uav;
                gripper;
        }

        Banned {
                toAltitude == "medium";
                toAltitude == "high";
        }
```

```
        NLSlots {
                default: toLocation;
        }

        Body
        {
                do {
                    intend transport(fromLocation = THIS.fromLocation,
                                     toLocation = THIS.toLocation,
                                     carryObject = THIS.carryObject,
                                     toSpeed = THIS.toSpeed,
                                     toAltitude = THIS.toAltitude,
                                     fromSpeed = THIS.fromSpeed,
                                     fromAltitude = THIS.fromAltitude);
                } while(still_fire(toLocation));
        }
}
```

References

[ABD+01] James F. Allen, Donna K. Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. Toward conversational human-computer interaction. *AI Magazine*, 22(4):27–37, 2001.

[ACD+02] Abderrahammane Aggoun, David Chan, Pierre Dufresene, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Kish Shen, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. ECLiPSe user manual release. Technical report, International Computer Limited and Imperial College London, May 2002.

[act] History of mobile robots, http://www.activrobots.com/history/, 2002.

[ASF+95] James F. Allen, Lenhart K. Schubert, George Ferguson, Peter Heeman, Chung Hee Hwang, Tsuneaki Kato, Marc Light, Nathaniel G. Martin, Bradford W. Miller, Massimo Poesio, and David R. Traum. The TRAINS project: A case study in building a conversational planning agent. *Journal of Experimental and Theoretical AI*, 7:7–48, 1995.

[BL01] Edward Bachelder and Nancy Leveson. Describing and probing complex system behavior: A graphical approach. In *Proceedings of the Aviation Safety Conference*, Sept 2001.

[Bla01] Nate Blaylock. Retroactive recognition of interleaved plans for natural language dialogue. Technical report, Department of Computer Science, University of Rochester, December 2001.

[Bra90] Michael E. Bratman. *What Is Intention*, chapter 2, pages 15–31. In Cohen et al. [CMP90], 1990.

[Cla96] Herbert H. Clark. *Using Language*. Cambridge University Press, 1996.

[CMP90] Philip R. Cohen, Jerry Morgan, and Martha E. Pollack, editors. *Intentions in Communication*. MIT Press, Cambridge, MA, 1990.

[DGA+93] John Dowding, Jean Mark Gawron, Doug Appelt, John Bear, Lynn Cherny anRobert Moore, and Douglas Moran. GEMINI: a natural language system for spoken-language understanding. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, 1993.

[DGK+00] Patrick Doherty, Gösta Granlund, Krzystof Kuchcinski, Erik Sandewall, Klas Nordberg, Erik Skarman, and Johan Wiklund. The WITAS unmanned aerial vehicle project. In *European Conference on Artificial Intelligence (ECAI 2000)*, 2000.

[GK96] Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.

[GK98] B. Grosz and S. Kraus. The evolution of SharedPlans, 1998.

[GS90] Barbara J. Grosz and Candace L. Sidner. *Plans for Discourse*, chapter 20, pages 417–444. In Cohen et al. [CMP90], 1990.

[hon] Honda robot, http://world.honda.com/robot/, 2002.

[jtp] Jtp: An object-oriented modular reasoning system http://ksl.stanford.edu/software/jtp/, 2002.

[KBM98] David Kortenkamp, R Bonasso, and R Murphy, editors. *AI-Based Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press, 1998.

[Kon97] Kurt Konolige. COLBERT: A language for reactive control in sapphira. In *KI - Kunstliche Intelligenz*, pages 31–52, 1997.

[LA90] Diane J. Litman and James F. Allen. *Discourse Processing and Commonsense Plans*, chapter 17, pages 365–388. In Cohen et al. [CMP90], 1990.

[LBPA02] Markus Lockelt, Tilman Becker, Norbert Pfleger, and Jan Alexandersson. Making sense of partial. In *Proceedings of the sixth workshop on the semantics and pragmatics of dialogue (EDILOG 2002)*, pages 101–107, september 2002.

[Lev00] Nancy Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. on Software Engineering*, January 2000.

[LGP02] Oliver Lemon, Alexander Gruenstein, and Stanley Peters. Collaborative activities and multi-tasking in dialogue systems. *Traitment automatique des langues*, 43(2):131–154, 2002. Special issue on dialogue.

[Loc94] Karen E. Lochbaum. *Using Collaborative Plans to Model the Intentional Structure of Discourse*. PhD thesis, Harvard University, Cambridge, MA, 1994.

[LRGB01] Ian Lewin, Manny Rayner, Genevieve Gorrell, and Johan Boye. Plug and play speech understanding. In *Proceeedings of second SIGdial workshop on discourse and dialogue*, 2001.

[Mye96] Karen L. Myers. A procedural knowledge approach to task-level control. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 158–165. AAAI Press, 1996.

[NSSS02] Stefano Novello, Joachim Schimpf, Kish Shen, and Josh Singer. ECLiPSe embedding and interfacing manual. Technical report, IC-Parc and Parc Technologies Limited, May 2002.

[Par00] Terence Parr. *ANTLR Reference Manual*. jGuru.com, October 2000.

[Pol90] Martha E. Pollack. *Plans as Complex Mental Attitudes*, chapter 5, pages 77–103. In Cohen et al. [CMP90], 1990.

[RHJ00] Manny Rayner, Beth Ann Hockey, and Frankie James. A compact architecture for dialogue management based on scripts and meta-outputs. In *Proceedings of Applied Natural Language Processing (ANLP)*, 2000.

[RLR+02] Jeff Rickel, Neal Lesh, Charles Rich, Candace L. Sidner, and Abigail Gertner. Collaborative discourse theory as a foundation for tutorial dialogue. In Springer-Verlag, editor, *Proceedings of Sixth International Conference on Intelligent Tutorial Systems*, pages 542 – 551, june 2002.

[RSL01] Charles Rich, Candace L. Sidner, and Neal Lesh. COLLAGEN: Applying collaborative discourse theory to human-computer interaction. *AI Magazine, Special Issue on Intelligent User Interfaces*, 2001.

[SdOB99] Janienke Sturm, Els dens Os, and Lou Boves. Dialogue management in the Dutch ARISE train timetable information system. In *Proceedings of the 5th international conference on speech communication and technology (EUROSPEECH)*, 1999.

[SKR95] Alessandro Saffiotti, Kurt Konolige, and Enrique H. Ruspini. A multivalued-logic approach to integrating planning and control. *Artificial Intelligence*, 76(1-2):481–526, 1995.

[SP00] S. Seneff and J. Polifroni. Dialogue management in the mercury flight reservation system. In *Proceedings ANLP/NAACL 2000 Workshop on Conversational Systems*, 2000.

[Ste01] Amanda J. Stent. *Dialogue Systems as Conversational Partners: Applying conversation acts theory to natural language generation for task-oriented mixed-initiative spoken dialogue.* PhD thesis, University of Rochester, August 2001.

[TA94] David Traum and James Allen. Towards a formal theory of repair in plan execution and plan recognition. *Procedings of UK planning and scheduling special interest group*, 1994.

[WNS97] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, 1997.